API Documentation

API Documentation

June 16, 2009

Contents

1 Package networkx

NetworkX

NetworkX (NX) is a Python package for the creation, manipulation, and study of the structure, dynamics, and functions of complex networks.

https://networkx.lanl.gov/

Using

Just write in Python

```
>>> import networkx as NX
>>> G=NX.Graph()
>>> G.add_edge(1,2)
>>> G.add_node("spam")
>>> print G.nodes()
[1, 2, 'spam']
>>> print G.edges()
[(1, 2)]
```

Graph classes Graph

A simple graph that has no self-loops or multiple (parallel) edges.

An empty graph is created with

```
>>> G=Graph()
```

DiGraph

A directed graph that has no self-loops or multiple (parallel) edges. Subclass of Graph.

An empty digraph is created with

```
>>> G=DiGraph()
```

XGraph

A graph that has (optional) self-loops or multiple (parallel) edges and arbitrary data on the edges. Subclass of Graph.

An empty graph is created with

```
>>> G=XGraph()
```

XDiGraph

A directed graph that has (optional) self-loops or multiple (parallel) edges and arbitrary data on the edges.

A simple digraph that has no self-loops or multiple (parallel) edges. Subclass of DiGraph which is a subclass of Graph.

An empty digraph is created with

```
>>> G=DiGraph()
```

The XGraph and XDiGraph classes extend the Graph and DiGraph classes by allowing (optional) self loops, multiedges and by decorating each edge with an object x.

Each XDiGraph or XGraph edge is a 3-tuple e=(n1,n2,x), representing an edge between nodes n1 and n2 that is decorated with the object x. Here n1 and n2 are (hashable) node objects and x is a (not necessarily hashable) edge object. If multiedges are allowed, G.get_edge(n1,n2) returns a list of edge objects.

Whether an XGraph or XDiGraph allow self-loops or multiple edges is determined initially via parameters selfloops=True/False and multiedges=True/False. For example, the example empty XGraph created above is equivalent to

```
>>> G=XGraph(selfloops=False, multiedges=False)
```

Similar defaults hold for XDiGraph. The command

```
>>> G=XDiGraph(multiedges=True)
```

creates an empty digraph G that does not allow selfloops but does allow for multiple (parallel) edges. Methods exist for allowing or disallowing each feature after instatiation as well.

Note that if G is an XGraph then G.add_edge(n1,n2) will add the edge (n1,n2,None), and G.delete_edge(n1,n2) will attempt to delete the edge (n1,n2,None). In the case of multiple edges between nodes n1 and n2, one can use G.delete_multiedge(n1,n2) to delete all edges between n1 and n2.

Notation The following shorthand is used throughout NetworkX documentation and code: (we use mathematical notation n,v,w,... to indicate a node, v=vertex=node).

G,G1,G2,H,etc: Graphs

n,n1,n2,u,v,v1,v2: nodes (vertices)

nlist: a list of nodes (vertices)

nbunch: a "bunch" of nodes (vertices). An nbunch is either a single node of the graph or any iterable container/iterator of nodes. The distinction is determined by checking if nbunch is in the graph. If you use iterable containers as nodes you should be careful when using nbunch.

e=(n1,n2): an edge (a python "2-tuple"), also written n1-n2 (if undirected) and n1->n2 (if directed).

e=(n1,n2,x): an edge triple ("3-tuple") containing the two nodes connected and the edge data/label/object stored associated with the edge. The object x, or a list of objects (if multiedges=True), can be obtained using G.get_edge(n1,n2)

elist: a list of edges (as 2- or 3-tuples)

ebunch: a bunch of edges (as 2- or 3-tuples). An ebunch is any iterable (non-string) container of edge-tuples (either 2-tuples, 3-tuples or a mixture).

Warning:

- The ordering of objects within an arbitrary nbunch/ebunch can be machine-dependent.
- Algorithms should treat an arbitrary nbunch/ebunch as once-through-and-exhausted iterable containers.
- len(nbunch) and len(ebunch) need not be defined.

Methods Each class provides basic graph methods.

Mutating Graph methods

- G.add_node(n), G.add_nodes_from(nlist)
- G.delete_node(n), G.delete_nodes_from(nlist)
- G.add_edge(n1,n2), G.add_edge(e), where e=(u,v)
- G.add_edges_from(ebunch)
- G.delete_edge(n1,n2), G.delete_edge(e), where e=(u,v)
- G.delete_edges_from(ebunch)
- G.add_path(nlist)
- G.add_cycle(nlist)
- G.clear()
- G.subgraph(nbunch,inplace=True)

Non-mutating Graph methods

- len(G)
- G.has_node(n)
- n in G (equivalent to G.has_node(n))
- for n in G: (iterate through the nodes of G)
- G.nodes()
- G.nodes_iter()
- G.has_edge(n1,n2), G.has_neighbor(n1,n2), G.get_edge(n1,n2)

- G.edges(), G.edges(n), G.edges(nbunch)
- G.edges_iter(), G.edges_iter(n), G.edges_iter(nbunch)
- G.neighbors(n)
- G[n] (equivalent to G.neighbors(n))
- G.neighbors_iter(n) # iterator over neighbors
- G.number_of_nodes(), G.order()
- G.number_of_edges(), G.size()
- G.edge_boundary(nbunch1), G.node_boundary(nbunch1)
- G.degree(n), G.degree(nbunch)
- G.degree_iter(n), G.degree_iter(nbunch)
- G.is_directed()
- G.info() # print variaous info about a graph
- G.prepare_nbunch(nbunch) # return list of nodes in G and nbunch

Methods returning a new graph

- G.subgraph(nbunch)
- G.subgraph(nbunch,create_using=H)
- G.copy()
- G.to_undirected()
- G.to_directed()

Implementation Notes The graph classes implement graphs using data structures based on an adjacency list implemented as a node-centric dictionary of dictionaries. The dictionary contains keys corresponding to the nodes and the values are dictionaries of neighboring node keys with the value None (the Python None type) for Graph and DiGraph or user specified (default is None) for XGraph and XDiGraph. The dictionary of dictionary structure allows fast addition, deletion and lookup of nodes and neighbors in large graphs.

Similarities between XGraph and Graph XGraph and Graph differ in the way edge data is handled. XGraph edges are 3-tuples (n1,n2,x) and Graph edges are 2-tuples (n1,n2). XGraph inherits from the Graph class, and XDiGraph from the DiGraph class.

Graph and XGraph are similar in the following ways:

1. Edgeless graphs are the same in XGraph and Graph. For an edgeless graph, represented by G (member of the Graph class) and XG (member of XGraph class), there is no difference between the datastructures G.adj and XG.adj, other than possibly in the ordering of the keys in the adj dict.

2. Basic graph construction code for G=Graph() will also work for G=XGraph(). In the Graph class, the simplest graph construction consists of a graph creation command G=Graph() followed by a list of graph construction commands, consisting of successive calls to the methods:

   G.add_node, G.add_nodes_from, G.add_edge, G.add_edges, G.add_path, G.add_cycle G.delete_node, G.delete_nodes_from, G.delete_edge, G.delete_edges_from

   with all edges specified as 2-tuples,

   If one replaces the graph creation command with G=XGraph(), and then apply the identical list of construction commands, the resulting XGraph object will be a simple graph G with identical datastructure G.adj. This property ensures reuse of code developed for graph generation in the Graph class.

Version: 0.36

Date: Tue Jun 16 14:09:53 2009

Author: Aric Hagberg <hagberg@lanl.gov> Dan Schult <dschult@colgate.edu> Pieter Swart <swart@lanl.gov>

License: LGPL

1.1 Modules

  - **leda**: Read graphs in LEDA format.
    (Section 41, p. 151)
  - **nx_yaml**: Read and write NetworkX graphs in YAML format.
    (Section 42, p. 152)
  - **sparsegraph6**: Read graphs in graph6 and sparse6 format.
    (Section 43, p. 153)
- **release**: Release data for NetworkX.
  (Section 44, p. 155)
- **search**: Search algorithms.
  (Section 45, p. 156)
- **spectrum**: Laplacian, adjacency matrix, and spectrum of graphs.
  (Section 46, p. 158)
- **tests** *(Section 47, p. 160)*
  - **benchmark** *(Section 48, p. 161)*
  - **drawing** *(Section 49, p. 163)*
  - **generators** *(Section 50, p. 164)*
  - **readwrite** *(Section 51, p. 165)*
  - **test** *(Section 52, p. 166)*
- **threshold**: Threshold Graphs - Creation, manipulation and identification.
  (Section 53, p. 167)
- **tree**: EXPERIMENTAL: Base classes for trees and forests.
  (Section 54, p. 175)
- **utils**: Utilities for networkx package
  (Section 55, p. 188)
- **xdigraph**: Base class for XDiGraph.
  (Section 56, p. 192)
- **xgraph**: Base class for XGraph.
  (Section 57, p. 205)

1.2 Variables

Name	Description
__package__	**Value:** `'networkx'`

2 Module networkx.centrality

Centrality measures. **Author:** Aric Hagberg (hagberg@lanl.gov) Pieter Swart (swart@lanl.gov) Sasha Gutfraind (ag362@cornell.edu)

2.1 Functions

brandes_betweenness_centrality(*G*, *normalized*=`True`, *weighted_edges*=`False`)

Compute the betweenness centrality for nodes in G: the fraction of number of shortests paths that pass through each node.

The keyword normalized (default=True) specifies whether the betweenness values are normalized by b=b/(n-1)(n-2) where n is the number of nodes in G.

The keyword weighted_edges (default=False) specifies whether to use edge weights (otherwise weights are all assumed equal).

The algorithm is from Ulrik Brandes, A Faster Algorithm for Betweenness Centrality. Journal of Mathematical Sociology 25(2):163-177, 2001.
http://www.inf.uni-konstanz.de/algo/publications/b-fabc-01.pdf

newman_betweenness_centrality(*G*, *v*=`None`, *cutoff*=`None`, *normalized*=`True`, *weighted_edges*=`False`)

"Load" centrality for nodes.

This actually computes 'load' and not betweenness. See
https://networkx.lanl.gov/ticket/103

The fraction of number of shortests paths that go through each node counted according to the algorithm in

Scientific collaboration networks: II. Shortest paths, weighted networks, and centrality, M. E. J. Newman, Phys. Rev. E 64, 016132 (2001).

Returns a dictionary of betweenness values keyed by node. The betweenness is normalized to be between [0,1].

If normalized=False the resulting betweenness is not normalized.

If weighted_edges is True then use Dijkstra for finding shortest paths.

betweenness_centrality(*G*, *normalized*=`True`, *weighted_edges*=`False`)

Compute the betweenness centrality for nodes in G: the fraction of number of shortests paths that pass through each node.

The keyword normalized (default=True) specifies whether the betweenness values are normalized by b=b/(n-1)(n-2) where n is the number of nodes in G.

The keyword weighted_edges (default=False) specifies whether to use edge weights (otherwise weights are all assumed equal).

The algorithm is from Ulrik Brandes, A Faster Algorithm for Betweenness Centrality. Journal of Mathematical Sociology 25(2):163-177, 2001.
http://www.inf.uni-konstanz.de/algo/publications/b-fabc-01.pdf

load_centrality(*G*, *v*=`None`, *cutoff*=`None`, *normalized*=`True`, *weighted_edges*=`False`)

"Load" centrality for nodes.

This actually computes 'load' and not betweenness. See
https://networkx.lanl.gov/ticket/103

The fraction of number of shortests paths that go through each node counted according to the algorithm in

Scientific collaboration networks: II. Shortest paths, weighted networks, and centrality, M. E. J. Newman, Phys. Rev. E 64, 016132 (2001).

Returns a dictionary of betweenness values keyed by node. The betweenness is normalized to be between [0,1].

If normalized=False the resulting betweenness is not normalized.

If weighted_edges is True then use Dijkstra for finding shortest paths.

betweenness_centrality_source(*G*, *normalized*=`True`, *weighted_edges*=`False`, *sources*=`None`)

Enchanced version of the method in centrality module that allows specifying a list of sources (subgraph).

weighted_edges:: consider edge weights by running Dijkstra's algorithm (no effect on unweighted graphs).

sources:: list of nodes to consider as subgraph

See Sec. 4 in Ulrik Brandes, A Faster Algorithm for Betweenness Centrality. Journal of Mathematical Sociology 25(2):163-177, 2001.
http://www.inf.uni-konstanz.de/algo/publications/b-fabc-01.pdf

This algorithm does not count the endpoints, i.e. a path from s to t does not contribute to the betweenness of s and t.

edge_betweenness(*G*, *normalized*=`True`, *weighted_edges*=`False`, *sources*=`None`)

Edge betweenness centrality.

weighted_edges:: consider edge weights by running Dijkstra's algorithm (no effect on unweighted graphs).

sources:: list of nodes to consider as subgraph

edge_load(*G*, *nodes*=`False`, *cutoff*=`False`)

Edge Betweenness

WARNING:

This module is for demonstration and testing purposes.

degree_centrality(*G*, *v*=`None`)

Degree centrality for nodes (fraction of nodes connected to).

Returns a dictionary of degree centrality values keyed by node.

The degree centrality is normalized to the maximum possible degree in the graph G.

closeness_centrality(*G*, *v*=`None`, *weighted_edges*=`False`)

Closeness centrality for nodes (1/average distance to all nodes).

Returns a dictionary of closeness centrality values keyed by node. The closeness centrality is normalized to be between 0 and 1.

2.2 Variables

Name	Description
__package__	**Value:** `'networkx'`

3 Module networkx.cliques

Cliques - Find and manipulate cliques of graphs

Note that finding the largest clique of a graph has been shown to be an NP complete problem so the algorithms here could take a LONG time to run. In practice it hasn't been too bad for the graphs tested. **Date:** $Date: 2005-06-15 07:56:03 -0600 (Wed, 15 Jun 2005) $

Author: Dan Schult (dschult@colgate.edu)

3.1 Functions

find_cliques(*G*)

Find_cliques algorithm based on Bron & Kerbosch

This algorithm searchs for maximal cliques in a graph. maximal cliques are the largest complete subgraph containing a given point. The largest maximal clique is sometimes called the maximum clique.

This algorithm produces the list of maximal cliques each of which are a list of the members of the clique.

Based on Algol algorithm published by Bron & Kerbosch A C version is available as part of the rambin package. http://www.ram.org/computing/rambin/rambin.html

Reference:

```
@article{362367,
 author = {Coen Bron and Joep Kerbosch},
 title = {Algorithm 457: finding all cliques of an undi-
rected graph},
 journal = {Commun. ACM},
 volume = {16},
 number = {9},
 year = {1973},
 issn = {0001-0782},
 pages = {575--577},
 doi = {http://doi.acm.org/10.1145/362342.362367},
 publisher = {ACM Press},
 }
```

make_max_clique_graph(*G*, *create_using*=None, *name*=None)

Create the maximal clique graph of a graph. It finds the maximal cliques and treats these as nodes. The nodes are connected if they have common members in the original graph. Theory has done a lot with clique graphs, but I haven't seen much on maximal clique graphs.

Note: This should be the same as make_clique_bipartite followed by project_up, but it saves all the intermediate stuff.

make_clique_bipartite(*G*, *fpos*=None, *create_using*=None, *name*=None)

Create a bipartite clique graph from a graph G. Nodes of G are retained as the "bottom nodes" of B and cliques of G become "top nodes" of B. Edges are present if a bottom node belongs to the clique represented by the top node.

Returns a Graph with additional attribute B.node_type which is "Bottom" or "Top" appropriately.

if fpos is not None, a second additional attribute B.pos is created to hold the position tuple of each node for viewing the bipartite graph.

project_down(*B*, *create_using*=None, *name*=None)

Project a bipartite graph B down onto its "Bottom Nodes". The nodes retain their names and are connected if they share a common Top Node in the Bipartite Graph. Returns a Graph.

project_up(*B*, *create_using*=None, *name*=None)

Project a bipartite graph B up onto its "Top Nodes". The nodes retain their names and are connected if they share a common Bottom Node in the Bipartite Graph. Returns a Graph.

graph_clique_number(*G*, *cliques*=None)

Return the clique number (size the largest clique) for G. Optional list of cliques can be input if already computed.

graph_number_of_cliques(*G*, *cliques*=None)

Returns the number of maximal cliques in G Optional list of cliques can be input if already computed.

node_clique_number(*G*, *nodes*=None, *with_labels*=False, *cliques*=None)

Returns the size of the largest maximal clique containing each given node.

Returns a single or list depending on input nodes. Returns a dict keyed by node if "with_labels=True". Optional list of cliques can be input if already computed.

number_of_cliques(*G*, *nodes*=None, *cliques*=None, *with_labels*=False)

Returns the number of maximal cliques for each node.

Returns a single or list depending on input nodes. Returns a dict keyed by node if "with_labels=True". Optional list of cliques can be input if already computed.

cliques_containing_node(*G*, *nodes*=None, *cliques*=None, *with_labels*=False)

Returns a list of cliques containing the given node.

Returns a single list or list of lists depending on input nodes. Returns a dict keyed by node if "with_labels=True". Optional list of cliques can be input if already computed.

3.2 Variables

Name	Description
__credits__	**Value:** ''
__revision__	**Value:** '$Revision: 1021 $'
__package__	**Value:** 'networkx'

4 Module networkx.cluster

Compute clustering coefficients and transitivity of graphs.

Clustering coefficient For each node find the fraction of possible triangles that are triangles, c_i = triangles_i / (k_i*(k_i-1)/2) where k_i is the degree of node i.

A coefficient for the whole graph is the average C = avg(c_i)

Transitivity Find the fraction of all possible triangles which are in fact triangles. Possible triangles are identified by the number of "triads" (two edges with a shared vertex)

T = 3*triangles/triads

Date: $Date: 2005-06-14 12:48:10 -0600 (Tue, 14 Jun 2005) $

Author: Aric Hagberg (hagberg@lanl.gov) Pieter Swart (swart@lanl.gov) Dan Schult (dschult@colgate.edu)

4.1 Functions

triangles(*G*, *nbunch*=`None`, *with_labels*=`False`)

Return number of triangles for nbunch of nodes. If nbunch is None, then return triangles for every node. If with_labels is True, return a dict keyed by node.

Note: Each triangle is counted three times: once at each vertex.

average_clustering(*G*)

Average clustering coefficient for a graph.

Note: this is a space saving routine; It might be faster to use clustering to get a list and then take average.

clustering(*G*, *nbunch*=`None`, *with_labels*=`False`, *weights*=`False`)

Clustering coefficient for each node in nbunch.

If with_labels is True, return a dict keyed by node.

If both with_labels and weights are True, return both a clustering coefficient dict keyed by node and a dict of weights based on degree. The weights are the fraction of connected triples in the graph which include the keyed node. Ths is useful in moving from transitivity for clustering coefficient and back.

transitivity(G)

Transitivity (fraction of transitive triangles) for a graph

4.2 Variables

Name	Description
__credits__	**Value:** `''`
__revision__	**Value:** `'$Revision: 1012 $'`
__package__	**Value:** `None`

5 Module networkx.component

Connected components and strongly connected components. **Author:** Eben Kennah (ekenah@t7.lanl.gov) Aric Hagberg (hagberg@lanl.gov)

5.1 Functions

connected_components(G)

Return a list of lists of nodes in each connected component of G.

The list is ordered from largest connected component to smallest. For undirected graphs only.

number_connected_components(G)

Return the number of connected components in G. For undirected graphs only.

is_connected(G)

Return True if G is connected. For undirected graphs only.

connected_component_subgraphs(G)

Return a list of graphs of each connected component of G. The list is ordered from largest connected component to smallest. For undirected graphs only.

For example, to get the largest connected component: >>> H=connected_component_subgraphs(G)[0]

node_connected_component(G, n)

Return a list of nodes of the connected component containing node n.

For undirected graphs only.

strongly_connected_components(*G*)

Returns list of strongly connected components in G. Uses Tarjan's algorithm with Nuutila's modifications. Nonrecursive version of algorithm.

References:

> R. Tarjan (1972). Depth-first search and linear graph algorithms. SIAM Journal of Computing 1(2):146-160.
>
> E. Nuutila and E. Soisalon-Soinen (1994). On finding the strongly connected components in a directed graph. Information Processing Letters 49(1): 9-14.

kosaraju_strongly_connected_components(*G*, *source*=`None`)

Returns list of strongly connected components in G. Uses Kosaraju's algorithm.

strongly_connected_components_recursive(*G*)

Returns list of strongly connected components in G. Uses Tarjan's algorithm with Nuutila's modifications. this recursive version of the algorithm will hit the Python stack limit for large graphs.

strongly_connected_component_subgraphs(*G*)

Return a list of graphs of each strongly connected component of G. The list is ordered from largest connected component to smallest.

For example, to get the largest strongly connected component: >>> H=strongly_connected_component_subgraphs(G)[0]

number_strongly_connected_components(*G*)

Return the number of connected components in G. For undirected graphs only.

is_strongly_connected(*G*)

Return True if G is strongly connected.

5.2 Variables

Name	Description
__revision__	**Value:** ' '

continued on next page

Name	Description
__package__	**Value:** `'networkx'`

6 Module networkx.convert

Convert NetworkX graphs to and from other formats.

from_whatever attemps to guess the input format

Create a 10 node random digraph

```
>>> from networkx import *
>>> import numpy
>>> a=numpy.reshape(numpy.random.random_integers(0,1,size=100),(10,10))
>>> D=from_whatever(a,create_using=DiGraph()) # or D=DiGraph(a)
```

For graphviz formats see networkx.drawing.nx_pygraphviz or networkx.drawing.nx_pydot.

$Id: convert.py 701 2007-11-08 05:08:53Z aric $ **Author:** Aric Hagberg (hagberg@lanl.gov)

6.1 Functions

from_whatever(*thing*, *create_using*=None)

Attempt to make a NetworkX graph from an known type.

Current known types are:

any NetworkX graph dict-of-dicts dist-of-lists numpy matrix numpy ndarray scipy sparse matrix pygraphviz agraph

to_dict_of_lists(*G*, *nodelist*=None)

Return graph G as a Python dict of lists.

If nodelist is defined return a dict of lists with only those nodes.

Completely ignores edge data for XGraph and XDiGraph.

from_dict_of_lists(*d*, *create_using*=None)

Return a NetworkX graph G from a Python dict of lists.

to_dict_of_dicts(*G*, *nodelist*=None, *edge_data*=None)

Return graph G as a Python dictionary of dictionaries.

If nodelist is defined return a dict of dicts with only those nodes.

If edge_data is given, the value of the dictionary will be set to edge_data for all edges. This is useful to make an adjacency matrix type representation with 1 as the edge data.

from_dict_of_dicts(*d*, *create_using*=None)

Return a NetworkX graph G from a Python dictionary of dictionaries.

The value of the inner dict becomes the edge_data for the NetworkX graph EVEN if create_using is a NetworkX Graph which doesn't ever use this data.

If create_using is an XGraph/XDiGraph with multiedges==True, the edge_data should be a list, though this routine does not check for that.

to_numpy_matrix(*G*, *nodelist*=None)

Return adjacency matrix of graph as a numpy matrix.

If nodelist is defined return adjacency matrix with nodes in nodelist in the order specified. If not the ordering is whatever order the method G.nodes() produces.

For Graph/DiGraph types which have no edge data The value of the entry A[u,v] is one if there is an edge u-v and zero otherwise.

For XGraph/XDiGraph the edge data is assumed to be a weight and be able to be converted to a valid numpy type (e.g. an int or a float). The value of the entry A[u,v] is the weight given by get_edge(u,v) one if there is an edge u-v and zero otherwise.

Graphs with multi-edges are not handled.

from_numpy_matrix(*A*, *create_using*=None)

Return networkx graph G from numpy matrix adjacency list.

```
>>> G=from_numpy_matrix(A)
```

to_scipy_sparse_matrix(*G*, *nodelist*=None)

Return adjacency matrix of graph as a scipy sparse matrix.

Uses lil_matrix format. To convert to other formats see scipy.sparse documentation.

If nodelist is defined return adjacency matrix with nodes in nodelist in the order specified. If not the ordering is whatever order the method G.nodes() produces.

For Graph/DiGraph types which have no edge data The value of the entry A[u,v] is one if there is an edge u-v and zero otherwise.

For XGraph/XDiGraph the edge data is assumed to be a weight and be able to be converted to a valid numpy type (e.g. an int or a float). The value of the entry A[u,v] is the weight given by get_edge(u,v) one if there is an edge u-v and zero otherwise.

Graphs with multi-edges are not handled.

```
>>> A=scipy_sparse_matrix(G)
>>> A.tocsr() # convert to compressed row storage
```

from_scipy_sparse_matrix(*A*, *create_using*=None)

Return networkx graph G from scipy scipy sparse matrix adjacency list.

```
>>> G=from_scipy_sparse_matrix(A)
```

6.2 Variables

Name	Description
__package__	**Value:** 'networkx'

7 Module networkx.cores

Find and manipulate the k-cores of a graph **Date:** $Date: 2005-03-30 16:56:28 -0700 (Wed, 30 Mar 2005) $

Author: Dan Schult(dschult@colgate.edu)

7.1 Functions

find_cores(*G*, *with_labels*=`True`)

Return the core number for each vertex.

See: arXiv:cs.DS/0310049 by Batagelj and Zaversnik

If with_labels is True a dict is returned keyed by node to the core number. If with_labels is False a list of the core numbers is returned.

7.2 Variables

Name	Description
__credits__	**Value:** `''`
__revision__	**Value:** `'$Revision: 911 $'`
__package__	**Value:** `None`

8 Module networkx.dag

Algorithms for directed acyclic graphs (DAGs). **Author:** Aric Hagberg (hagberg@lanl.gov) Dan Schult(dschult@colgate.edu)

8.1 Functions

is_directed_acyclic_graph(G)

Return True if the graph G is a directed acyclic graph (DAG).

Otherwise return False.

topological_sort(G)

Return a list of nodes of the digraph G in topological sort order.

A topological sort is a nonunique permutation of the nodes such that an edge from u to v implies that u appears before v in the topological sort order.

If G is not a directed acyclic graph no topological sort exists and the Python keyword None is returned.

This algorithm is based on a description and proof at
http://www2.toki.or.id/book/AlgDesignManual/book/book2/node70.htm

See also is_directed_acyclic_graph()

topological_sort_recursive(G)

Return a list of nodes of the digraph G in topological sort order.

This is a recursive version of topological sort.

8.2 Variables

Name	Description
__revision__	**Value:** `''`
__package__	**Value:** `'networkx'`

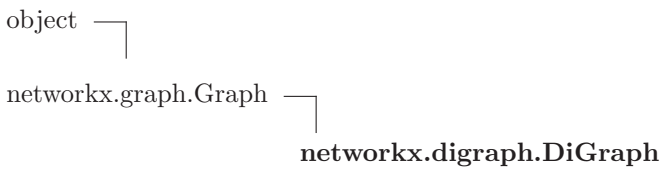
9 Module networkx.digraph

Base class for digraphs. **Author:** Aric Hagberg (hagberg@lanl.gov) Pieter Swart (swart@lanl.gov) Dan Schult(dschult@colgate.edu)

9.1 Variables

Name	Description
__package__	**Value:** `'networkx'`

9.2 Class DiGraph



Known Subclasses: networkx.xdigraph.XDiGraph, networkx.tree.DirectedForest, networkx.tree.DirectedTree

A graph with directed edges. Subclass of Graph.

DiGraph inherits from Graph, overriding the following methods:

- __init__: replaces self.adj with the dicts self.pred and self.succ
- add_node
- delete_node
- add_edge
- delete_edge
- add_nodes_from
- delete_nodes_from
- add_edges_from
- delete_edges_from
- edges_iter
- degree_iter
- copy
- clear
- subgraph
- is_directed

- to_directed
- to_undirected

Digraph adds the following methods to those of Graph:

- successors
- successors_iter
- predecessors
- predecessors_iter
- out_degree
- out_degree_iter
- in_degree
- in_degree_iter

9.2.1 Methods

__init__(*self*, *data*=None, *name*='')

Initialize Graph.

```
>>> G=Graph(name="empty")
```

creates empty graph G with G.name="empty" Overrides: object.__init__ extit(inherited documentation)

add_node(*self*, *n*)

Add a single node to the digraph.

The node n can be any hashable object except None.

A hashable object is one that can be used as a key in a Python dictionary. This includes strings, numbers, tuples of strings and numbers, etc. On many platforms this also includes mutables such as Graphs, though one should be careful that the hash doesn't change on mutables.

```
>>> from networkx import *
>>> G=DiGraph()
>>> K3=complete_graph(3)
>>> G.add_nodes_from(K3)     # add the nodes from K3 to G
>>> G.nodes()
[0, 1, 2]
>>> G.clear()
>>> G.add_node(K3)           # add the graph K3 as a node in G.
>>> G.number_of_nodes()
1
```

Overrides: networkx.graph.Graph.add_node

add_nodes_from(*self*, *nlist*)

Add multiple nodes to the digraph.

nlist: A container of nodes that will be iterated through once (thus it should be an iterator or be iterable). Each element of the container should be a valid node type: any hashable type except None. See add_node for details. Overrides: networkx.graph.Graph.add_nodes_from

delete_node(*self*, *n*)

Delete node n from the digraph. Attempting to delete a non-existent node will raise a NetworkXError. Overrides: networkx.graph.Graph.delete_node

delete_nodes_from(*self*, *nlist*)

Remove nodes in nlist from the digraph.

nlist: an iterable or iterator containing valid node names.

Attempting to delete a non-existent node will raise an exception. This could mean some nodes in nlist were deleted and some valid nodes were not! Overrides: networkx.graph.Graph.delete_nodes_from

add_edge(*self*, *u*, *v*=None)

Add a single directed edge (u,v) to the digraph.

>> G.add_edge(u,v) and >>> G.add_edge((u,v)) are equivalent forms of adding a single edge between nodes u and v. The nodes u and v will be automatically added if not already in the graph. They must be a hashable (except None) Python object.

For example, the following examples all add the edge (1,2) to the digraph G.

```
>>> G=DiGraph()
>>> G.add_edge( 1, 2 )           # explicit two node form
>>> G.add_edge( (1,2) )          # single edge as tuple of two nodes
>>> G.add_edges_from( [(1,2)] ) # list of edges form
```

Overrides: networkx.graph.Graph.add_edge

add_edges_from(*self*, *ebunch*)

Add all the edges in ebunch to the graph.

ebunch: Container of 2-tuples (u,v). The container must be iterable or an iterator. It is iterated over once. Adding the same edge twice has no effect and does not raise an exception.

See add_edge for an example. Overrides: networkx.graph.Graph.add_edges_from

delete_edge(*self*, *u*, *v*=None)

Delete the single directed edge (u,v) from the digraph.

Can be used in two basic forms >>> G.delete_edge(u,v) and G.delete_edge((u,v)) are equivalent ways of deleting a directed edge u->v.

If the edge does not exist return without complaining. Overrides: networkx.graph.Graph.delete_edge

delete_edges_from(*self*, *ebunch*)

Delete the directed edges in ebunch from the digraph.

ebunch: Container of 2-tuples (u,v). The container must be iterable or an iterator. It is iterated over once.

Edges that are not in the digraph are ignored. Overrides: networkx.graph.Graph.delete_edges_from

out_edges_iter(*self*, *nbunch*=None)

Return iterator that iterates once over each edge pointing out of nodes in nbunch, or over all edges in digraph if no nodes are specified.

See edges() for definition of nbunch.

Nodes in nbunch that are not in the graph will be (quietly) ignored.

in_edges_iter(*self*, *nbunch*=None)

Return iterator that iterates once over each edge adjacent to nodes in nbunch, or over all edges in digraph if no nodes are specified.

See edges() for definition of nbunch.

Nodes in nbunch that are not in the graph will be (quietly) ignored.

edges_iter(*self*, *nbunch*=None)

Return iterator that iterates once over each edge pointing out of nodes in nbunch, or over all edges in digraph if no nodes are specified.

See edges() for definition of nbunch.

Nodes in nbunch that are not in the graph will be (quietly) ignored. Overrides: networkx.graph.Graph.edges_iter

out_edges(*self*, *nbunch*=None)

Return list of all edges that point out of nodes in nbunch, or a list of all edges in graph if no nodes are specified.

See edges() for definition of nbunch.

Nodes in nbunch that are not in the graph will be (quietly) ignored.

in_edges(*self*, *nbunch*=None)

Return list of all edges that point in to nodes in nbunch, or a list of all edges in graph if no nodes are specified.

See edges() for definition of nbunch.

Nodes in nbunch that are not in the graph will be (quietly) ignored.

successors_iter(*self*, *n*)

Return an iterator for successor nodes of n.

predecessors_iter(*self*, *n*)

Return an iterator for predecessor nodes of n.

successors(*self*, *n*)

Return sucessor nodes of n.

predecessors(*self*, *n*)

Return predecessor nodes of n.

out_neighbors(*self*, *n*)

Return sucessor nodes of n.

in_neighbors(*self*, *n*)

Return predecessor nodes of n.

neighbors(*self*, *n*)

Return sucessor nodes of n. Overrides: networkx.graph.Graph.neighbors

neighbors_iter(*self*, *n*)

Return an iterator for successor nodes of n. Overrides: networkx.graph.Graph.neighbors_iter

degree_iter(*self*, *nbunch*=`None`, *with_labels*=`False`)

Return iterator that returns in_degree(n)+out_degree(n) or (n,in_degree(n)+out_degree(n)) for all n in nbunch. If nbunch is ommitted, then iterate over *all* nodes.

Can be called in three ways: G.degree_iter(n): return iterator the degree of node n G.degree_iter(nbunch): return a list of values, one for each n in nbunch (nbunch is any iterable container of nodes.) G.degree_iter(): same as nbunch = all nodes in graph.

If with_labels=True, iterator will return an (n,in_degree(n)+out_degree(n)) tuple of node and degree.

Any nodes in nbunch but not in the graph will be (quietly) ignored. Overrides: networkx.graph.Graph.degree_iter

in_degree_iter(*self*, *nbunch*=`None`, *with_labels*=`False`)

Return iterator for in_degree(n) or (n,in_degree(n)) for all n in nbunch.

If nbunch is ommitted, then iterate over *all* nodes.

See degree_iter method for more details.

out_degree_iter(*self*, *nbunch*=`None`, *with_labels*=`False`)

Return iterator for out_degree(n) or (n,out_degree(n)) for all n in nbunch.

If nbunch is ommitted, then iterate over *all* nodes.

See degree_iter method for more details.

out_degree(*self*, *nbunch*=`None`, *with_labels*=`False`)

Return out-degree of single node or of nbunch of nodes.

If nbunch is omitted or nbunch=None, then return out-degrees of *all* nodes.

in_degree(*self*, *nbunch*=`None`, *with_labels*=`False`)

Return in-degree of single node or of nbunch of nodes.

If nbunch is omitted or nbunch=None, then return in-degrees of *all* nodes.

clear(*self*)

Remove name and delete all nodes and edges from digraph. Overrides: networkx.graph.Graph.clear

copy(*self*)

Return a (shallow) copy of the digraph.

Identical to dict.copy() of adjacency dicts pred and succ, with name copied as well. Overrides: networkx.graph.Graph.copy

subgraph(*self*, *nbunch*, *inplace*=`False`, *create_using*=`None`)

Return the subgraph induced on nodes in nbunch.

nbunch: can be a single node or any iterable container of of nodes. (It can be an iterable or an iterator, e.g. a list, set, graph, file, numeric array, etc.)

Setting inplace=True will return the induced subgraph in original graph by deleting nodes not in nbunch. This overrides create_using. Warning: this can destroy the graph.

Unless otherwise specified, return a new graph of the same type as self. Use (optional) create_using=R to return the resulting subgraph in R. R can be an existing graph-like object (to be emptied) or R can be a call to a graph object, e.g. create_using=DiGraph(). See documentation for empty_graph()

Note: use subgraph(G) rather than G.subgraph() to access the more general subgraph() function from the operators module. Overrides: networkx.graph.Graph.subgraph

is_directed(*self*)

Return True if a directed graph. Overrides: networkx.graph.Graph.is_directed

to_undirected(*self*)

Return the undirected representation of the digraph.

A new graph is returned (the underlying graph). The edge u-v is in the underlying graph if either u->v or v->u is in the digraph. Overrides: networkx.graph.Graph.to_undirected

to_directed(*self*)

Return a directed representation of the digraph.

This is already directed, so merely return a copy. Overrides: networkx.graph.Graph.to_directed

reverse(*self*)

Return a new digraph with the same vertices and edges as G but with the directions of the edges reversed.

Inherited from networkx.graph.Graph(Section 28.2)

__contains__(), __getitem__(), __iter__(), __len__(), __str__(), add_cycle(), add_path(), degree(), edge_boundary(), edges(), get_edge(), has_edge(), has_neighbor(), has_node(), info(), node_boundary(), nodes(), nodes_iter(), number_of_edges(), number_of_nodes(), order(), prepare_nbunch(), size()

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __new__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __subclasshook__()

9.2.2 Properties

Name	Description
Inherited from object	
__class__	

10 Module networkx.distance

Shortest paths, diameter, radius, eccentricity, and related methods. **Author:** Aric Hagberg (hagberg@lanl.gov) Dan Schult(dschult@colgate.edu)

10.1 Functions

eccentricity(*G*, *v*=`None`, *sp*=`None`, *with_labels*=`False`)

Return the eccentricity of node v in G (or all nodes if v is None).

The eccentricity is the maximum of shortest paths to all other nodes.

The optional keyword sp must be a dict of dicts of shortest_path_length keyed by source and target. That is, sp[v][t] is the length from v to t.

If with_labels=True return dict of eccentricities keyed by vertex.

diameter(*G*, *e*=`None`)

Return the diameter of the graph G.

The diameter is the maximum of all pairs shortest path.

periphery(*G*, *e*=`None`)

Return the periphery of the graph G.

The periphery is the set of nodes with eccentricity equal to the diameter.

radius(*G*, *e*=`None`)

Return the radius of the graph G.

The radius is the minimum of all pairs shortest path.

center(*G*, *e*=None)

Return the center of graph G.

The center is the set of nodes with eccentricity equal to radius.

10.2 Variables

Name	Description
__package__	**Value:** 'networkx'

11 Package networkx.drawing

11.1 Modules

- **layout**: Layout (positioning) algorithms for graph drawing.
  (Section 12, p. 37)
- **nx_agraph**: Interface to pygraphviz AGraph class.
  (Section 13, p. 39)
- **nx_pydot**: Import and export networkx networks to dot format using pydot.
  (Section 14, p. 42)
- **nx_pylab**: Draw networks with matplotlib (pylab).
  (Section 15, p. 44)
- **nx_vtk**: Draw networks in 3d with vtk.
  (Section 16, p. 50)

11.2 Variables

Name	Description
__package__	**Value:** `'networkx.drawing'`

12 Module networkx.drawing.layout

Layout (positioning) algorithms for graph drawing. **Date:** $Date: 2005-06-15 08:53:26 -0600 (Wed, 15 Jun 2005) $

Author: Aric Hagberg (hagberg@lanl.gov) Dan Schult(dschult@colgate.edu)

12.1 Functions

circular_layout(*G*, *dim*=2)

Circular layout.

Crude version that doesn't try to minimize edge crossings.

shell_layout(*G*, *nlist*=None, *dim*=2)

Shell layout. Crude version that doesn't try to minimize edge crossings.

nlist is an optional list of lists of nodes to be drawn at each shell level. Only one shell with all nodes will be drawn if not specified.

random_layout(*G*, *dim*=2)

Random layout.

spring_layout(*G*, *iterations*=50, *dim*=2, *node_pos*=None)

Spring force model layout

spectral_layout(*G*, *dim*=2, *vpos*=None, *iterations*=1000, *eps*=0.001)

Return the position vectors for drawing G using spectral layout.

graph_low_ev_pi(*uhat*, *G*, *eps*=0.001, *iterations*=10000)

Power Iteration method to find smallest eigenvectors of Laplacian(G). Note: constant eigenvector has eigenvalue=0 but is not included in the count of smallest eigenvalues.

uhat -- list of p initial guesses (dicts) for the p eigenvectors. G -- The Graph from which Laplacian is calculated. eps -- tolerance for norm of change in eigenvalue estimate. iterations -- maximum number of iterations to use.

12.2 Variables

Name	Description
__credits__	**Value:** ''
__revision__	**Value:** '$Revision: 1033 $'
__package__	**Value:** 'networkx.drawing'
__warningregistry__	**Value:** {('Not importing directory \'/usr/lib/python2.6/dist-pack...

13 Module networkx.drawing.nx_agraph

Interface to pygraphviz AGraph class.

Usage

```
>>> from networkx import *
>>> G=complete_graph(5)
>>> A=to_agraph(G)
>>> H=from_agraph(A)
```

Author: Aric Hagberg (hagberg@lanl.gov)

13.1 Functions

from_agraph(*A*, *create_using*=None)

Return a NetworkX XGraph or XDiGraph from a pygraphviz graph.

```
>>> X=from_agraph(A)
```

The XGraph X will have a dictionary X.graph_attr containing the default graphviz attributes for graphs, nodes and edges.

Default node attributes will be in the dictionary X.node_attr which is keyed by node.

Edge attributes will be returned as edge data in the graph X.

If you want a Graph with no attributes attached instead of an XGraph with attributes use

```
>>> G=Graph(X)
```

to_agraph(*N*, *graph_attr*=None, *node_attr*=None, *edge_attr*=None, *strict*=True)

Return a pygraphviz graph from a NetworkX graph N.

If N is a Graph or DiGraph, graphviz attributes can be supplied through the arguments

graph_attr: dictionary with default attributes for graph, nodes, and edges
keyed by 'graph', 'node', and 'edge' to attribute dictionaries

node_attr: dictionary keyed by node to node attribute dictionary

edge_attr: dictionary keyed by edge tuple to edge attribute dictionary

If N is an XGraph or XDiGraph an attempt will be made first to copy properties attached to the graph (see from_agraph) and then updated with the calling arguments if any.

write_dot(*G*, *path*)

Write NetworkX graph G to Graphviz dot format on path.

Path can be a string or a file handle.

read_dot(*path*, *create_using*=None)

Return a NetworkX XGraph or XdiGraph from a dot file on path.

Path can be a string or a file handle.

graphviz_layout(*G*, *prog*='neato', *root*=None, *args*='')

Create layout using graphviz. Returns a dictionary of positions keyed by node.

```
>>> from networkx import *
>>> G=petersen_graph()
>>> pos=graphviz_layout(G)
>>> pos=graphviz_layout(G,prog='dot')
```

This is a wrapper for pygraphviz_layout.

pygraphviz_layout(*G*, *prog*='neato', *root*=None, *args*='')

Create layout using pygraphviz and graphviz. Returns a dictionary of positions keyed by node.

```
>>> from networkx import *
>>> G=petersen_graph()
>>> pos=pygraphviz_layout(G)
>>> pos=pygraphviz_layout(G,prog='dot')
```

13.2 Variables

Name	Description
__package__	**Value:** 'networkx.drawing'

14 Module networkx.drawing.nx_pydot

Import and export networkx networks to dot format using pydot.

Provides:

- write_dot()
- read_dot()
- graphviz_layout()
- pydot_layout()
- pydot_from_networkx()
- networkx_from_pydot()

Either this module or nx_pygraphviz can be used to interface with graphviz.

References:

- pydot Homepage: http://www.dkbza.org/pydot.html
- Graphviz: http://www.research.att.com/sw/tools/graphviz/
- DOT Language: http://www.research.att.com/~erg/graphviz/info/lang.html

Date: $Date: 2005-06-15 08:55:33 -0600 (Wed, 15 Jun 2005) $

Author: Aric Hagberg (hagberg@lanl.gov)

14.1 Functions

write_dot(*G*, *path*=`False`)

Write G to a graphviz dot file.

read_dot(*path*=`False`)

Creates an networkx graph from a dot file

pydot_from_networkx(*N*)

Creates a pydot graph from an networkx graph N

networkx_from_pydot(*D*, *create_using*=None)

Creates an networkx graph from an pydot graph D

graphviz_layout(*G*, *prog*='neato', *root*=None, ***kwds*)

Create layout using pydot and graphviz. Returns a dictionary of positions keyed by node.

```
>>> pos=graphviz_layout(G)
>>> pos=graphviz_layout(G,prog='dot')
```

This is a wrapper for pydot_layout.

pydot_layout(*G*, *prog*='neato', *root*=None, ***kwds*)

Create layout using pydot and graphviz. Returns a dictionary of positions keyed by node.

```
>>> pos=pydot_layout(G)
>>> pos=pydot_layout(G,prog='dot')
```

14.2 Variables

Name	Description
__credits__	**Value:** """\n"""
__revision__	**Value:** "$Revision: 1034 $"

15 Module networkx.drawing.nx_pylab

Draw networks with matplotlib (pylab).

Provides:

- draw()
- draw_networkx()
- draw_networkx_nodes()
- draw_networkx_edges()
- draw_networkx_labels()
- draw_circular
- draw_random
- draw_spectral
- draw_spring
- draw_shell
- draw_graphviz

References:

- matplotlib: http://matplotlib.sourceforge.net/
- pygraphviz: http://networkx.lanl.gov/pygraphviz/

Date: $Date: 2005-06-15 11:29:39 -0600 (Wed, 15 Jun 2005) $

Author: Aric Hagberg (hagberg@lanl.gov)

15.1 Functions

draw(*G*, *pos*=None, *ax*=None, *hold*=None, ***kwds*)

Draw the graph G with matplotlib (pylab).

This is a pylab friendly function that will use the current pylab figure axes (e.g. subplot).

pos is a dictionary keyed by vertex with a two-tuple of x-y positions as the value. See networkx.layout for functions that compute node positions.

Usage:

```
>>> from networkx import *
>>> G=dodecahedral_graph()
>>> draw(G)
>>> pos=graphviz_layout(G)
>>> draw(G,pos)
>>> draw(G,pos=spring_layout(G))
```

Also see doc/examples/draw_*

for more see pylab.scatter

NB: this has the same name as pylab.draw so beware when using

```
>>> from networkx import *
```

since you will overwrite the pylab.draw function.

A good alternative is to use

```
>>> import pylab as P
>>> import networkx as NX
>>> G=NX.dodecahedral_graph()
```

and then use

```
>>> NX.draw(G)  # networkx draw()
```

and >>> P.draw() # pylab draw() **Parameters**

nodelist:	list of nodes to be drawn (default=G.nodes())
edgelist:	list of edges to be drawn (default=G.edges())
node_size:	scalar or array of the same length as nodelist (default=300)
node_color:	single color string or numeric/numarray array of floats (default='r')
node_shape:	node shape (default='o'), or 'so^>v<dph8' see pylab.scatter
alpha:	transparency (default=1.0)
cmap:	colormap for mapping intensities (default=None)

draw_networkx(*G*, *pos*, *with_labels*=`True`, ***kwds*)

Draw the graph G with given node positions pos

Usage:

```
>>> from networkx import *
>>> import pylab as P
>>> ax=P.subplot(111)
>>> G=dodecahedral_graph()
>>> pos=spring_layout(G)
>>> draw_networkx(G,pos,ax=ax)
```

This is same as 'draw' but the node positions *must* be specified in the variable pos. pos is a dictionary keyed by vertex with a two-tuple of x-y positions as the value. See networkx.layout for functions that compute node positions.

An optional matplotlib axis can be provided through the optional keyword ax.

with_labels contols text labeling of the nodes

Also see:

draw_networkx_nodes() draw_networkx_edges() draw_networkx_labels()

draw_networkx_nodes(*G*, *pos*, *nodelist*=`None`, *node_size*=`300`, *node_color*=`'r'`, *node_shape*=`'o'`, *alpha*=`1.0`, *cmap*=`None`, *vmin*=`None`, *vmax*=`None`, *ax*=`None`, ***kwds*)

Draw nodes of graph G

This draws only the nodes of the graph G.

pos is a dictionary keyed by vertex with a two-tuple of x-y positions as the value. See networkx.layout for functions that compute node positions.

nodelist is an optional list of nodes in G to be drawn. If provided only the nodes in nodelist will be drawn.

see draw_networkx for the list of other optional parameters.

draw_networkx_edges(*G*, *pos*, *edgelist*=None, *width*=1.0, *edge_color*='k', *style*='solid', *alpha*=1.0, *edge_cmap*=None, *edge_vmin*=None, *edge_vmax*=None, *ax*=None, *arrows*=True, ***kwds*)

Draw the edges of the graph G

This draws only the edges of the graph G.

pos is a dictionary keyed by vertex with a two-tuple of x-y positions as the value. See networkx.layout for functions that compute node positions.

edgelist is an optional list of the edges in G to be drawn. If provided, only the edges in edgelist will be drawn.

For directed graphs, "arrows" (actually just thicker stubs) are drawn at the head end. Arrows can be turned off with keyword arrows=False.

See draw_networkx for the list of other optional parameters.

draw_networkx_labels(*G*, *pos*, *labels*=None, *font_size*=12, *font_color*='k', *font_family*='sans-serif', *font_weight*='normal', *alpha*=1.0, *ax*=None, ***kwds*)

Draw node labels on the graph G

pos is a dictionary keyed by vertex with a two-tuple of x-y positions as the value. See networkx.layout for functions that compute node positions.

labels is an optional dictionary keyed by vertex with node labels as the values. If provided only labels for the keys in the dictionary are drawn.

See draw_networkx for the list of other optional parameters.

draw_circular(*G*, ***kwargs*)

Draw the graph G with a circular layout

draw_random(*G*, ***kwargs*)

Draw the graph G with a random layout.

draw_spectral(*G*, ***kwargs*)

Draw the graph G with a spectral layout.

draw_spring(*G*, ***kwargs*)

Draw the graph G with a spring layout

draw_shell(*G*, ***kwargs*)

Draw networkx graph with shell layout

draw_graphviz(*G*, *prog*=`'neato'`, ***kwargs*)

Draw networkx graph with graphviz layout

draw_nx(*G*, *pos*, ***kwds*)

For backward compatibility; use draw or draw_networkx

15.2 Variables

Name	Description
__credits__	**Value:** `''`
__revision__	**Value:** `'$Id'`
__package__	**Value:** `'networkx.drawing'`

16 Module networkx.drawing.nx_vtk

Draw networks in 3d with vtk.

References:

- vtk: http://www.vtk.org/

Date: $Date: 2005-06-17 08:10:29 -0600 (Fri, 17 Jun 2005) $

Author: Aric Hagberg (hagberg@lanl.gov)

16.1 Functions

draw_nxvtk(*G*, *node_pos*)

Draw networkx graph in 3d with nodes at node_pos.

See layout.py for functions that compute node positions.

node_pos is a dictionary keyed by vertex with a three-tuple of x-y positions as the value.

The node color is plum. The edge color is banana.

All the nodes are the same size.

16.2 Variables

Name	Description
__credits__	**Value:** `''`
__revision__	**Value:** `'$Revision: 1051 $'`
__package__	**Value:** `'networkx.drawing'`

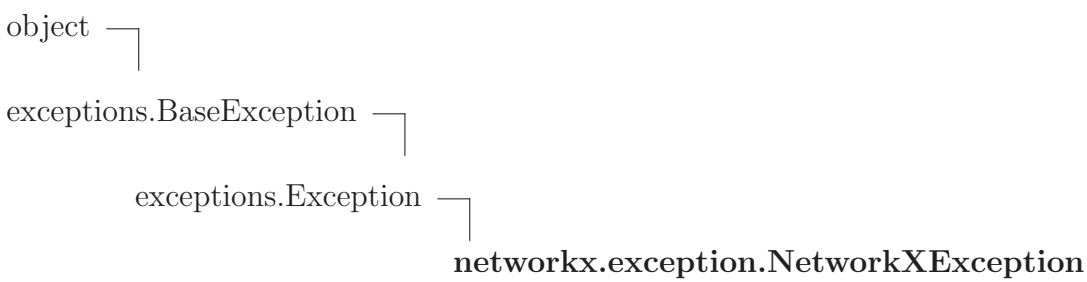
17 Module networkx.exception

Base exceptions and errors for NetworkX. **Author:** Aric Hagberg (hagberg@lanl.gov) Pieter Swart (swart@lanl.gov) Dan Schult(dschult@colgate.edu)

17.1 Variables

Name	Description
__package__	**Value:** `None`

17.2 Class NetworkXException



Known Subclasses: networkx.exception.NetworkXError

Base class for exceptions in NetworkX.

17.2.1 Methods

Inherited from exceptions.Exception

__init__(), __new__()

Inherited from exceptions.BaseException

__delattr__(), __getattribute__(), __getitem__(), __getslice__(), __reduce__(), __repr__(), __setattr__(), __setstate__(), __str__(), __unicode__()

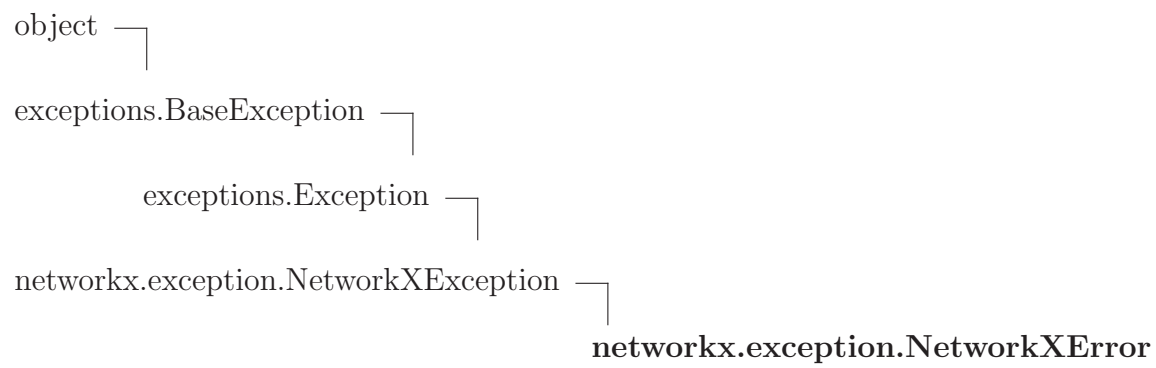
Inherited from object

__format__(), __hash__(), __reduce_ex__(), __sizeof__(), __subclasshook__()

17.2.2 Properties

Name	Description
Inherited from exceptions.BaseException	
args, message	
Inherited from object	
__class__	

17.3 Class NetworkXError



Exception for a serious error in NetworkX

17.3.1 Methods

Inherited from exceptions.Exception

__init__(), __new__()

Inherited from exceptions.BaseException

__delattr__(), __getattribute__(), __getitem__(), __getslice__(), __reduce__(), __repr__(), __setattr__(), __setstate__(), __str__(), __unicode__()

Inherited from object

__format__(), __hash__(), __reduce_ex__(), __sizeof__(), __subclasshook__()

17.3.2 Properties

Name	Description
Inherited from exceptions.BaseException	
args, message	
Inherited from object	

continued on next page

Name	Description
__class__	

18 Module networkx.function

Functional interface to graph properties. **Author:** Aric Hagberg (hagberg@lanl.gov) Pieter Swart (swart@lanl.gov) Dan Schult(dschult@colgate.edu)

18.1 Functions

nodes(*G*)

Return a copy of the graph nodes in a list.

nodes_iter(*G*)

Return an iterator over the graph nodes.

edges(*G*, *nbunch*=`None`)

Return list of edges adjacent to nodes in nbunch.

Return all edges if nbunch is unspecified or nbunch=None.

For digraphs, edges=out_edges

edges_iter(*G*, *nbunch*=`None`)

Return iterator over edges adjacent to nodes in nbunch.

Return all edges if nbunch is unspecified or nbunch=None.

For digraphs, edges=out_edges

degree(*G*, *nbunch*=`None`, *with_labels*=`False`)

Return degree of single node or of nbunch of nodes. If nbunch is ommitted, then return degrees of *all* nodes.

neighbors(*G*, *n*)

Return a list of nodes connected to node n.

number_of_nodes(*G*)

Return the order of a graph = number of nodes.

number_of_edges(*G*)

Return the size of a graph = number of edges.

density(*G*)

Return the density of a graph.

density = size/(order*(order-1)/2) density()=0.0 for an edge-less graph and 1.0 for a complete graph.

degree_histogram(*G*)

Return a list of the frequency of each degree value.

The degree values are the index in the list. Note: the bins are width one, hence len(list) can be large (Order(number_of_edges))

is_directed(*G*)

Return True if graph is directed.

18.2 Variables

Name	Description
__package__	**Value:** `None`

19 Package networkx.generators

A package for generating various graphs in networkx.

19.1 Modules

- **atlas**: Generators for the small graph atlas.
  (Section 20, p. 57)
- **bipartite**: Generators and functions for bipartite graphs.
  (Section 21, p. 58)
- **classic**: Generators for some classic graphs.
  (Section 22, p. 62)
- **degree_seq**: Generate graphs with a given degree sequence or expected degree sequence.
  (Section 23, p. 68)
- **directed**: Generators for some directed graphs.
  (Section 24, p. 78)
- **geometric**: Generators for geometric graphs.
  (Section 25, p. 81)
- **random_graphs**: Generators for random graphs
  (Section 26, p. 82)
- **small**: Various small and named graphs, together with some compact generators.
  (Section 27, p. 91)

19.2 Variables

Name	Description
__package__	**Value:** `'networkx.generators'`

20 Module networkx.generators.atlas

Generators for the small graph atlas.

See "An Atlas of Graphs" by Ronald C. Read and Robin J. Wilson, Oxford University Press, 1998.

Because of its size, this module is not imported by default. **Date:** $Date: 2005-03-30 16:56:28 -0700 (Wed, 30 Mar 2005) $

Author: Pieter Swart (swart@lanl.gov)

20.1 Functions

graph_atlas_g()

Return the list [G0,G1,...,G1252] of graphs as named in the Graph Atlas. G0,G1,...,G1252 are all graphs with up to 7 nodes.

The graphs are listed:

1. in increasing order of number of nodes;
2. for a fixed number of nodes, in increasing order of the number of edges;
3. for fixed numbers of nodes and edges, in increasing order of the degree sequence, for example 111223 < 112222;
4. for fixed degree sequence, in increasing number of automorphisms.

Note that indexing is set up so that for GAG=graph_atlas_g(), then G123=GAG[123] and G[0]=empty_graph(0)

20.2 Variables

Name	Description
__credits__	**Value:** `''`
__revision__	**Value:** `'$Revision: 911 $'`
__package__	**Value:** `'networkx.generators'`

21 Module networkx.generators.bipartite

Generators and functions for bipartite graphs. **Author:** Aric Hagberg (hagberg@lanl.gov) Pieter Swart (swart@lanl.gov) Dan Schult (dschult@colgate.edu)

21.1 Functions

bipartite_configuration_model(*aseq*, *bseq*, *create_using*=`None`, *seed*=`None`)

Return a random bipartite graph from two given degree sequences.

Nodes from the set A are connected to nodes in the set B by choosing randomly from the possible free stubs, one in A and one in B.

The sum of the two sequences must be equal: sum(aseq)=sum(bseq)

If no graph type is specified use XGraph with parallel edges.

If you want a graph with no parallel edges use create_using=Graph() but then the resulting degree sequences might not be exact. **Parameters**

`aseq`: degree sequence for node set A

`bseq`: degree sequence for node set B

bipartite_havel_hakimi_graph(*aseq*, *bseq*, *create_using*=`None`)

Return a bipartite graph from two given degree sequences using a Havel-Hakimi style construction.

Nodes from the set A are connected to nodes in the set B by connecting the highest degree nodes in set A to the highest degree nodes in set B until all stubs are connected.

The sum of the two sequences must be equal: sum(aseq)=sum(bseq)
Parameters

`aseq`: degree sequence for node set A

`bseq`: degree sequence for node set B

bipartite_reverse_havel_hakimi_graph(*aseq*, *bseq*, *create_using*=None)

Return a bipartite graph from two given degree sequences using a "reverse" Havel-Hakimi style construction.

Nodes from the set A are connected to nodes in the set B by connecting the highest degree nodes in set A to the lowest degree nodes in set B until all stubs are connected.

The sum of the two sequences must be equal: sum(aseq)=sum(bseq)

Parameters

`aseq`: degree sequence for node set A

`bseq`: degree sequence for node set B

bipartite_alternating_havel_hakimi_graph(*aseq*, *bseq*, *create_using*=None)

Return a bipartite graph from two given degree sequences using a alternating Havel-Hakimi style construction.

Nodes from the set A are connected to nodes in the set B by connecting the highest degree nodes in set A to alternatively the highest and the lowest degree nodes in set B until all stubs are connected.

The sum of the two sequences must be equal: sum(aseq)=sum(bseq)

Parameters

`aseq`: degree sequence for node set A

`bseq`: degree sequence for node set B

bipartite_preferential_attachment_graph(*aseq*, *p*, *create_using*=None)

Create a bipartite graph with a preferential attachment model from a given single "top" degree sequence.

Reference:

```
@article{guillaume-2004-bipartite,
  author = {Jean-Loup Guillaume and Matthieu Latapy},
  title = {Bipartite structure of all complex networks},
  journal = {Inf. Process. Lett.},
  volume = {90},
  number = {5},
  year = {2004},
  issn = {0020-0190},
  pages = {215--221},
  doi = {http://dx.doi.org/10.1016/j.ipl.2004.03.007},
  publisher = {Elsevier North-Holland, Inc.},
  address = {Amsterdam, The Netherlands, The Nether-
lands},
  }
```

Parameters

`aseq`: degree sequence for node set A (top)

`p`: probability that a new bottom node is added

bipartite_random_regular_graph(*d*, *n*, *create_using*=None)

UNTESTED:Generate a random bipartite graph of n nodes each with degree d.

Restrictions on n and d:

- n must be even
- n>=2*d

Nodes are numbered 0...n-1.

Algorithm inspired by random_regular_graph()

project(*B*, *nodes*, *create_using*=`None`)

Returns a graph that is the projection of the bipartite graph B onto the set of nodes given in list nodes.

The nodes retain their names and are connected if they share a common node in the node set of {B not nodes }.

No attempt is made to verify that the input graph B is bipartite.

bipartite_color(*G*)

is_bipartite(*G*)

Returns True if graph G is bipartite, False if not.

Traverse the graph G with depth-first-search and color nodes.

bipartite_sets(*G*)

Returns (X,Y) where X and Y are the nodes in each bipartite set of graph G. Fails with an error if graph is not bipartite.

21.2 Variables

Name	Description
__package__	**Value:** `'networkx.generators'`

22 Module networkx.generators.classic

Generators for some classic graphs.

The typical graph generator is called as follows:

```
>>> G=complete_graph(100)
```

returning the complete graph on n nodes labeled 0,..,99 as a simple graph. Except for empty_graph, all the generators in this module return a Graph class (i.e. a simple, undirected graph). **Date:** $Date: 2005-06-17 14:06:03 -0600 (Fri, 17 Jun 2005) $

Author: Aric Hagberg (hagberg@lanl.gov) Pieter Swart (swart@lanl.gov)

22.1 Functions

balanced_tree(r, h)

Return the perfectly balanced r-tree of height h.

For r>=2, h>=1, this is the rooted tree where all leaves are at distance h from the root. The root has degree r and all other internal nodes have degree r+1.

number_of_nodes = 1+r+r**2+...+r**h = (r**(h+1)-1)/(r-1),
number_of_edges = number_of_nodes - 1.

Node labels are the integers 0 (the root) up to number_of_nodes - 1.

barbell_graph(*m1*, *m2*)

Return the Barbell Graph: two complete graphs connected by a path.

For m1 > 1 and m2 >= 0.

Two identical complete graphs K_{m1} form the left and right bells, and are connected by a path P_{m2}.

The 2*m1+m2 nodes are numbered 0,...,m1-1 for the left barbell, m1,...,m1+m2-1 for the path, and m1+m2,...,2*m1+m2-1 for the right barbell.

The 3 subgraphs are joined via the edges (m1-1,m1) and (m1+m2-1,m1+m2). If m2=0, this is merely two complete graphs joined together.

This graph is an extremal example in David Aldous and Jim Fill's etext on Random Walks on Graphs.

complete_graph(*n*, *create_using*=`None`)

Return the Complete graph K_n with n nodes.

Node labels are the integers 0 to n-1.

complete_bipartite_graph(*n1*, *n2*)

Return the complete bipartite graph K_{n1_n2}.

Composed of two partitions with n1 nodes in the first and n2 nodes in the second. Each node in the first is connected to each node in the second.

Node labels are the integers 0 to n1+n2-1

circular_ladder_graph(*n*)

Return the circular ladder graph CL_n of length n.

CL_n consists of two concentric n-cycles in which each of the n pairs of concentric nodes are joined by an edge.

Node labels are the integers 0 to n-1

cycle_graph(*n*, *create_using*=None)

Return the cycle graph C_n over n nodes.

C_n is the n-path with two end-nodes connected.

Node labels are the integers 0 to n-1 If create_using is a DiGraph, the direction is in increasing order.

dorogovtsev_goltsev_mendes_graph(*n*)

Return the hierarchically constructed Dorogovtsev-Goltsev-Mendes graph.

n is the generation. See: arXiv:/cond-mat/0112143 by Dorogovtsev, Goltsev and Mendes.

empty_graph(*n*=0, *create_using*=None)

Return the empty graph with n nodes and zero edges.

Node labels are the integers 0 to n-1

For example: >>> from networkx import * >>> G=empty_graph(10) >>> G.number_of_nodes() 10 >>> G.number_of_edges() 0

The variable create_using should point to a "graph"-like object that will be cleaned (nodes and edges will be removed) and refitted as an empty "graph" with n nodes with integer labels. This capability is useful for specifying the class-nature of the resulting empty "graph" (i.e. Graph, DiGraph, MyWeirdGraphClass, etc.).

The variable create_using has two main uses: Firstly, the variable create_using can be used to create an empty digraph, network,etc. For example,

```
>>> n=10
>>> G=empty_graph(n,create_using=DiGraph())
```

will create an empty digraph on n nodes.

Secondly, one can pass an existing graph (digraph, pseudograph, etc.) via create_using. For example, if G is an existing graph (resp. digraph, pseudograph, etc.), then empty_graph(n,create_using=G) will empty G (i.e. delete all nodes and edges using G.clear() in base) and then add n nodes and zero edges, and return the modified graph (resp. digraph, pseudograph, etc.).

See also create_empty_copy(G).

grid_2d_graph(*m*, *n*, *periodic*=False)

Return the 2d grid graph of mxn nodes, each connected to its nearest neighbors. Optional argument periodic=True will connect boundary nodes via periodic boundary conditions.

grid_graph(*dim*, *periodic*=`False`)

Return the n-dimensional grid graph.

The dimension is the length of the list 'dim' and the size in each dimension is the value of the list element.

E.g. G=grid_graph(dim=[2,3]) produces a 2x3 grid graph.

If periodic=True then join grid edges with periodic boundary conditions.

hypercube_graph(*n*)

Return the n-dimensional hypercube.

Node labels are the integers 0 to 2**n - 1.

ladder_graph(*n*)

Return the Ladder graph of length n.

This is two rows of n nodes, with each pair connected by a single edge.

Node labels are the integers 0 to 2*n - 1.

lollipop_graph(*m*, *n*)

Return the Lollipop Graph; K_m connected to P_n.

This is the Barbell Graph without the right barbell.

For m>1 and n>=0, the complete graph K_m is connected to the path P_n. The resulting m+n nodes are labelled 0,...,m-1 for the complete graph and m,...,m+n-1 for the path. The 2 subgraphs are joined via the edge (m-1,m). If n=0, this is merely a complete graph.

Node labels are the integers 0 to number_of_nodes - 1.

(This graph is an extremal example in David Aldous and Jim Fill's etext on Random Walks on Graphs.)

null_graph(*create_using*=None)

Return the Null graph with no nodes or edges.

See empty_graph for the use of create_using.

path_graph(*n*, *create_using*=None)

Return the Path graph P_n of n nodes linearly connected by n-1 edges.

Node labels are the integers 0 to n - 1. If create_using is a DiGraph then the edges are directed in increasing order.

star_graph(*n*)

Return the Star graph with n+1 nodes: one center node, connected to n outer nodes.

Node labels are the integers 0 to n.

trivial_graph()

Return the Trivial graph with one node (with integer label 0) and no edges.

wheel_graph(*n*)

Return the wheel graph: a single hub node connected to each node of the (n-1)-node cycle graph.

Node labels are the integers 0 to n - 1.

22.2 Variables

Name	Description
__credits__	**Value:** ''
__revision__	**Value:** '$Revision: 1056 $'
__package__	**Value:** 'networkx.generators'

23 Module networkx.generators.degree_seq

Generate graphs with a given degree sequence or expected degree sequence. **Author:** Aric Hagberg (hagberg@lanl.gov) Pieter Swart (swart@lanl.gov) Dan Schult (dschult@colgate.edu)

23.1 Functions

configuration_model(*deg_sequence*, *seed*=None)

Return a random pseudograph with the given degree sequence.

- **deg_sequence: degree sequence, a list of integers with each entry** corresponding to the degree of a node (need not be sorted). A non-graphical degree sequence (i.e. one not realizable by some simple graph) will raise an Exception.
- seed: seed for random number generator (default=None)

```
>>> z=create_degree_sequence(100,powerlaw_sequence)
>>> G=configuration_model(z)
```

The pseudograph G is a networkx.XGraph that allows multiple (parallel) edges between nodes and edges that connect nodes to themseves (self loops).

To remove self-loops:

```
>>> G.ban_selfloops()
```

To remove parallel edges:

```
>>> G.ban_multiedges()
```

Steps:

- Check if deg_sequence is a valid degree sequence.
- Create N nodes with stubs for attaching edges
- Randomly select two available stubs and connect them with an edge.

As described by Newman [newman-2003-structure].

Nodes are labeled 1,.., len(deg_sequence), corresponding to their position in deg_sequence.

This process can lead to duplicate edges and loops, and therefore returns a pseudograph type. You can remove the self-loops and parallel edges (see above) with the likely result of not getting the exat degree sequence specified. This "finite-size effect" decreases as the size of the graph increases.

References:

[newman-2003-structure] M.E.J. Newman, "The structure and function of complex networks", SIAM REVIEW 45-2, pp 167-256, 2003.

expected_degree_graph(*w*, *seed*=None)

Return a random graph G(w) with expected degrees given by w.

```
>>> z=[10 for i in range(100)]
>>> G=expected_degree_graph(z)
```

To remove self-loops:

```
>>> G.ban_selfloops()
```

Reference:

```
@Article{connected-components-2002,
  author =        {Fan Chung and L. Lu},
  title =         {Connected components in random graphs
  with given expected degree sequences},
  journal =       {Ann. Combinatorics},
  year =          {2002},
  volume =        {6},
  pages =         {125-145},
  }
```

Parameters

`w`: a list of expected degrees

`seed`: seed for random number generator (default=None)

havel_hakimi_graph(*deg_sequence*, *seed*=`None`)

Return a simple graph with given degree sequence, constructed using the Havel-Hakimi algorithm.

- `deg_sequence`: **degree sequence, a list of integers with each entry** corresponding to the degree of a node (need not be sorted). A non-graphical degree sequence (not sorted). A non-graphical degree sequence (i.e. one not realizable by some simple graph) raises an Exception.
- `seed`: seed for random number generator (default=None)

The Havel-Hakimi algorithm constructs a simple graph by successively connecting the node of highest degree to other nodes of highest degree, resorting remaining nodes by degree, and repeating the process. The resulting graph has a high degree-associativity. Nodes are labeled 1,.., len(deg_sequence), corresponding to their position in deg_sequence.

See Theorem 1.4 in [chartrand-graphs-1996]. This algorithm is also used in the function is_valid_degree_sequence.

References:

[**chartrand-graphs-1996**] **G. Chartrand and L. Lesniak, "Graphs and Digraphs",** Chapman and Hall/CRC, 1996.

degree_sequence_tree(*deg_sequence*)

Make a tree for the given degree sequence.

A tree has #nodes-#edges=1 so the degree sequence must have len(deg_sequence)-sum(deg_sequence)/2=1

is_valid_degree_sequence(*deg_sequence*)

Return True if deg_sequence is a valid sequence of integer degrees equal to the degree sequence of some simple graph.

- `deg_sequence`: **degree sequence, a list of integers with each entry** corresponding to the degree of a node (need not be sorted). A non-graphical degree sequence (i.e. one not realizable by some simple graph) will raise an exception.

See Theorem 1.4 in [chartrand-graphs-1996]. This algorithm is also used in havel_hakimi_graph()

References:

[chartrand-graphs-1996] G. Chartrand and L. Lesniak, "Graphs and Digraphs", Chapman and Hall/CRC, 1996.

create_degree_sequence(*n*, *sfunction*=`None`, *max_tries*=`50`, ***kwds*)

Attempt to create a valid degree sequence of length n using specified function sfunction(n,**kwds).

- `n`: length of degree sequence = number of nodes
- `sfunction`: **a function, called as "sfunction(n,**kwds)",** that returns a list of n real or integer values.
- `max_tries`: **max number of attempts at creating valid degree** sequence.

Repeatedly create a degree sequence by calling sfunction(n,**kwds) until achieving a valid degree sequence. If unsuccessful after max_tries attempts, raise an exception.

For examples of sfunctions that return sequences of random numbers, see networkx.Utils.

```
>>> from networkx.utils import *
>>> seq=create_degree_sequence(10,uniform_sequence)
```

double_edge_swap(*G*, *nswap*=1)

Attempt nswap double-edge swaps on the graph G.

Return count of successful swaps. The graph G is modified in place. A double-edge swap removes two randomly choseen edges u-v and x-y and creates the new edges u-x and v-y:

```
u--v            u  v
       becomes  |  |
x--y            x  y
```

If either the edge u-x or v-y already exist no swap is performed so the actual count of swapped edges is always <= nswap

Does not enforce any connectivity constraints.

connected_double_edge_swap(*G*, *nswap*=1)

Attempt nswap double-edge swaps on the graph G.

Returns count of successful swaps. Enforces connectivity. The graph G is modified in place.

A double-edge swap removes two randomly choseen edges u-v and x-y and creates the new edges u-x and v-y:

```
u--v            u  v
       becomes  |  |
x--y            x  y
```

If either the edge u-x or v-y already exist no swap is performed so the actual count of swapped edges is always <= nswap

The initial graph G must be connected and the resulting graph is connected.

Reference:

```
@misc{gkantsidis-03-markov,
 author = "C. Gkantsidis and M. Mihail and E. Zegura",
 title = "The Markov chain simulation method for generat-
ing connected
          power law random graphs",
 year = "2003",
 url = "http://citeseer.ist.psu.edu/gkantsidis03markov.html"
}
```

li_smax_graph(*degree_seq*)

Generates a graph based with a given degree sequence and maximizing the s-metric. Experimental implementation.

Maximum s-metrix means that high degree nodes are connected to high degree nodes.

- `degree_seq`**: degree sequence, a list of integers with each entry** corresponding to the degree of a node. A non-graphical degree sequence raises an Exception.

Reference:

```
@unpublished{li-2005,
 author = {Lun Li and David Alderson and Reiko Tanaka
          and John C. Doyle and Walter Willinger},
 title = {Towards a Theory of Scale-Free Graphs:
         Definition, Properties, and  Implications (Ex-
tended Version)},
 url = {http://arxiv.org/abs/cond-mat/0501169},
 year = {2005}
}
```

The algorithm:

```
STEP 0 - Initialization
A = {0}
B = {1, 2, 3, ..., n}
O = {(i; j), ..., (k, l),...} where i < j, i <= k < l and
        d_i * d_j >= d_k *d_l
wA = d_1
dB = sum(degrees)

STEP 1 - Link selection
(a) If |O| = 0 TERMINATE. Return graph A.
(b) Select element(s) (i, j) in O hav-
ing the largest d_i * d_j , if for
        any i or j ei-
ther w_i = 0 or w_j = 0 delete (i, j) from O
(c) If there are no elements selected go to (a).
(d) Select the link (i, j) hav-
ing the largest value w_i (where for each
        (i, j) w_i is the smaller of w_i and w_j ), and pro-
ceed to STEP 2.

STEP 2 - Link addition
Type 1: i in A and j in B.
        Add j to the graph A and re-
move it from the set B add a link
        (i, j) to the graph A. Update variables:
        wA = wA + d_j -2 and dB = dB - d_j
```

connected_smax_graph(*degree_seq*)

Not implemented.

s_metric(*G*)

Return the "s-Metric" of graph G: the sum of the product deg(u)*deg(v) for every edge u-v in G

Reference:

```
@unpublished{li-2005,
 author = {Lun Li and David Alderson and
          John C. Doyle and Walter Willinger},
 title = {Towards a Theory of Scale-Free Graphs:
          Definition, Properties, and  Implications (Ex-
tended Version)},
 url = {http://arxiv.org/abs/cond-mat/0501169},
 year = {2005}
 }
```

23.2 Variables

Name	Description
__package__	**Value:** `'networkx.generators'`

24 Module networkx.generators.directed

Generators for some directed graphs.

gn_graph: growing network gnc_graph: growing network with copying gnr_graph: growing network with redirection **Author:** Aric Hagberg (hagberg@lanl.gov)

24.1 Functions

gn_graph(*n*, *kernel*=<function <lambda> at 0x27d21b8>, *seed*=None)

Return the GN (growing network) digraph with n nodes.

The graph is built by adding nodes one at a time with a link to one previously added node. The target node for the link is chosen with probability based on degree. The default attachment kernel is a linear function of degree.

The graph is always a (directed) tree.

Example:

```
>>> D=gn_graph(10)        # the GN graph
>>> G=D.to_undirected()   # the undirected version
```

To specify an attachment kernel use the kernel keyword

```
>>> D=gn_graph(10,kernel=lambda x:x**1.5) # A_k=k^1.5
```

Reference:

```
    @article{krapivsky-2001-organization,
    title   = {Organization of Growing Random Networks},
    author  = {P. L. Krapivsky and S. Redner},
    journal = {Phys. Rev. E},
    volume  = {63},
    pages   = {066123},
    year    = {2001},
    }
```

gnr_graph(*n*, *p*, *seed*=None)

Return the GNR (growing network with redirection) digraph with n nodes and redirection probability p.

The graph is built by adding nodes one at a time with a link to one previously added node. The previous target node is chosen uniformly at random. With probabiliy p the link is instead "redirected" to the successor node of the target. The graph is always a (directed) tree.

Example:

```
>>> D=gnr_graph(10,0.5)  # the GNR graph
>>> G=D.to_undirected()  # the undirected version
```

Reference:

```
    @article{krapivsky-2001-organization,
    title   = {Organization of Growing Random Networks},
    author  = {P. L. Krapivsky and S. Redner},
    journal = {Phys. Rev. E},
    volume  = {63},
    pages   = {066123},
    year    = {2001},
    }
```

gnc_graph(*n*, *seed*=None)

Return the GNC (growing network with copying) digraph with n nodes.

The graph is built by adding nodes one at a time with a links to one previously added node (chosen uniformly at random) and to all of that node's successors.

Reference:

```
    @article{krapivsky-2005-network,
    title   = {Network Growth by Copying},
    author  = {P. L. Krapivsky and S. Redner},
    journal = {Phys. Rev. E},
    volume  = {71},
    pages   = {036118},
    year    = {2005},
    }
```

24.2 Variables

Name	Description
__package__	**Value:** `'networkx.generators'`

25 Module networkx.generators.geometric

Generators for geometric graphs. **Date:** $Date: 2005-06-15 12:44:45 -0600 (Wed, 15 Jun 2005) $

Author: Aric Hagberg (hagberg@lanl.gov) Pieter Swart (swart@lanl.gov)

25.1 Functions

random_geometric_graph(*n*, *radius*, *create_using*=`None`, *repel*=`0.0`, *verbose*=`False`, *dim*=`2`)

Random geometric graph in the unit cube

Returned Graph has added attribute G.pos which is a dict keyed by node to the position tuple for the node.

25.2 Variables

Name	Description
__credits__	**Value:** `''`
__revision__	**Value:** `'$Revision: 1038 $'`
__package__	**Value:** `'networkx.generators'`

26 Module networkx.generators.random_graphs

Generators for random graphs **Date:** $Date: 2005-06-17 08:06:22 -0600 (Fri, 17 Jun 2005) $

Author: Aric Hagberg (hagberg@lanl.gov) Pieter Swart (swart@lanl.gov) Dan Schult(dschult@colgate.edu

26.1 Functions

fast_gnp_random_graph(*n*, *p*, *seed*=`None`)

Return a random graph G_{n,p}.

The G_{n,p} graph choses each of the possible [n(n-1)]/2 edges with probability p.

Sometimes called Erdős-Rényi graph, or binomial graph.

This algorithm is O(n+m) where m is the expected number of edges m=p*n*(n-1)/2.

It should be faster than gnp_random_graph when p is small, and the expected number of edges is small, (sparse graph).

See:

Batagelj and Brandes, "Efficient generation of large random networks", Phys. Rev. E, 71, 036113, 2005. **Parameters**

`n:`	the number of nodes
`p:`	probability for edge creation
`seed:`	seed for random number generator (default=None)

gnp_random_graph(*n*, *p*, *seed*=`None`)

Return a random graph G_{n,p}.

Choses each of the possible [n(n-1)]/2 edges with probability p. This is the same as binomial_graph and erdos_renyi_graph.

Sometimes called Erdős-Rényi graph, or binomial graph.

This is an O(n^2) algorithm. For sparse graphs (small p) see fast_gnp_random_graph.

P. Erdős and A. Rényi, On Random Graphs, Publ. Math. 6, 290 (1959). E. N. Gilbert, Random Graphs, Ann. Math. Stat., 30, 1141 (1959). **Parameters**

`n:` the number of nodes

`p:` probability for edge creation

`seed:` seed for random number generator (default=None)

binomial_graph(*n*, *p*, *seed*=`None`)

Return a random graph G_{n,p}.

Choses each of the possible [n(n-1)]/2 edges with probability p. This is the same as binomial_graph and erdos_renyi_graph.

Sometimes called Erdős-Rényi graph, or binomial graph.

This is an O(n^2) algorithm. For sparse graphs (small p) see fast_gnp_random_graph.

P. Erdős and A. Rényi, On Random Graphs, Publ. Math. 6, 290 (1959). E. N. Gilbert, Random Graphs, Ann. Math. Stat., 30, 1141 (1959). **Parameters**

`n:` the number of nodes

`p:` probability for edge creation

`seed:` seed for random number generator (default=None)

erdos_renyi_graph(*n*, *p*, *seed*=`None`)

Return a random graph G_{n,p}.

Choses each of the possible [n(n-1)]/2 edges with probability p. This is the same as binomial_graph and erdos_renyi_graph.

Sometimes called Erdős-Rényi graph, or binomial graph.

This is an O(n^2) algorithm. For sparse graphs (small p) see fast_gnp_random_graph.

P. Erdős and A. Rényi, On Random Graphs, Publ. Math. 6, 290 (1959). E. N. Gilbert, Random Graphs, Ann. Math. Stat., 30, 1141 (1959). **Parameters**

`n:` the number of nodes

`p:` probability for edge creation

`seed:` seed for random number generator (default=None)

dense_gnm_random_graph(*n*, *m*, *seed*=`None`)

Return the random graph G_{n,m}.

Gives a graph picked randomly out of the set of all graphs with n nodes and m edges. This algorithm should be faster than gnm_random_graph for dense graphs.

Algorithm by Keith M. Briggs Mar 31, 2006. Inspired by Knuth's Algorithm S (Selection sampling technique), in section 3.4.2 of

The Art of Computer Programming by Donald E. Knuth Volume 2 / Seminumerical algorithms Third Edition, Addison-Wesley, 1997.

Parameters

`n:` the number of nodes

`m:` the number of edges

`seed:` seed for random number generator (default=None)

gnm_random_graph(*n*, *m*, *seed*=`None`)

Return the random graph G_{n,m}.

Gives a graph picked randomly out of the set of all graphs with n nodes and m edges. **Parameters**

`n:` the number of nodes

`m:` the number of edges

`seed:` seed for random number generator (default=None)

newman_watts_strogatz_graph(*n*, *k*, *p*, *seed*=`None`)

Return a Newman-Watts-Strogatz small world graph.

First create a ring over n nodes. Then each node in the ring is connected with its k nearest neighbors. Then shortcuts are created by adding new edges as follows: for each edge u-v in the underlying "n-ring with k nearest neighbors"; with probability p add a new edge u-w with randomly-chosen existing node w. In contrast with watts_strogatz_graph(), no edges are removed. **Parameters**

`n:` the number of nodes

`k:` each node is connected to k nearest neighbors in ring topology

`p:` the probability of adding a new edge for each edge

`seed:` seed for random number generator (default=None)

watts_strogatz_graph(*n*, *k*, *p*, *seed*=`None`)

Return a Watts-Strogatz small world graph.

First create a ring over n nodes. Then each node in the ring is connected with its k nearest neighbors. Then shortcuts are created by rewiring existing edges as follows: for each edge u-v in the underlying "n-ring with k nearest neighbors"; with probability p replace u-v with a new edge u-w with randomly-chosen existing node w. In contrast with newman_watts_strogatz_graph(), the random rewiring does not increase the number of edges. **Parameters**

`n:` the number of nodes

`k:` each node is connected to k neighbors in the ring topology

`p:` the probability of rewiring an edge

`seed:` seed for random number generator (default=None)

random_regular_graph(*d*, *n*, *seed*=None)

Return a random regular graph of n nodes each with degree d, G_{n,d}.
Return False if unsuccessful.

n*d must be even

Nodes are numbered 0...n-1. To get a uniform sample from the space of random graphs you should chose d<nˆ{1/3}.

For algorith see Kim and Vu's paper.

Reference:

```
@inproceedings{kim-2003-generating,
author = {Jeong Han Kim and Van H. Vu},
title = {Generating random regular graphs},
booktitle = {Proceedings of the thirty-
fifth ACM symposium on Theory of computing},
year = {2003},
isbn = {1-58113-674-9},
pages = {213--222},
location = {San Diego, CA, USA},
doi = {http://doi.acm.org/10.1145/780542.780576},
publisher = {ACM Press},
}
```

The algorithm is based on an earlier paper:

```
@misc{ steger-1999-generating,
author = "A. Steger and N. Wormald",
title = "Generating random regular graphs quickly",
text = "Probability and Computing 8 (1999), 377-396.",
year = "1999",
url = "citeseer.ist.psu.edu/steger99generating.html",
}
```

barabasi_albert_graph(*n*, *m*, *seed*=None)

Return random graph using Barabási-Albert preferential attachment model.

A graph of n nodes is grown by attaching new nodes each with m edges that are preferentially attached to existing nodes with high degree.

The initialization is a graph with with m nodes and no edges.

Reference:

```
@article{barabasi-1999-emergence,
title   = {Emergence of scaling in random networks},
author  = {A. L. Barabási and R. Albert},
journal = {Science},
volume  = {286},
number  = {5439},
pages   = {509 -- 512},
year = {1999},
}
```

Parameters

`n`: the number of nodes

`m`: number of edges to attach from a new node to existing nodes

`seed`: seed for random number generator (default=None)

powerlaw_cluster_graph(*n*, *m*, *p*, *seed*=None)

Holme and Kim algorithm for growing graphs with powerlaw degree distribution and approximate average clustering.

Reference:

```
@Article{growing-holme-2002,
author =          {P. Holme and B. J. Kim},
title =   {Growing scale-
free networks with tunable clustering},
journal =         {Phys. Rev. E},
year =    {2002},
volume =          {65},
number =          {2},
pages =   {026107},
}
```

The average clustering has a hard time getting above a certain cutoff that depends on m. This cutoff is often quite low. Note that the transitivity (fraction of triangles to possible triangles) seems to go down with network size.

It is essentially the Barabási-Albert growth model with an extra step that each random edge is followed by a chance of making an edge to one of its neighbors too (and thus a triangle).

This algorithm improves on B-A in the sense that it enables a higher average clustering to be attained if desired.

It seems possible to have a disconnected graph with this algorithm since the initial m nodes may not be all linked to a new node on the first iteration like the BA model. **Parameters**

`n`: the number of nodes

`m`: the number of random edges to add for each new node

`p`: probability of adding a triangle after adding a random edge

`seed`: seed for random number generator (default=None)

random_lobster(*n*, *p1*, *p2*, *seed*=None)

Return a random lobster.

A caterpillar is a tree that reduces to a path graph when pruning all leaf nodes (p2=0). A lobster is a tree that reduces to a caterpillar when pruning all leaf nodes.

Parameters

`n`: the expected number of nodes in the backbone

`p1`: probability of adding an edge to the backbone

`p2`: probability of adding an edge one level beyond backbone

`seed`: seed for random number generator (default=None)

random_shell_graph(*constructor*, *seed*=None)

Return a random shell graph for the constructor given.

- constructor: a list of three-tuples [(n1,m1,d1),(n2,m2,d2),..] one for each shell, starting at the center shell.
- n : the number of nodes in the shell
- m : the number or edges in the shell
- **d (the ratio of inter (next) shell edges to intra shell edges.)**

  d=0 means no intra shell edges. d=1 for the last shell
- `seed`: seed for random number generator (default=None)

```
>>> constructor=[(10,20,0.8),(20,40,0.8)]
>>> G=random_shell_graph(constructor)
```

random_powerlaw_tree(*n*, *gamma*=3, *seed*=None, *tries*=100)

Return a tree with a powerlaw degree distribution.

A trial powerlaw degree sequence is chosen and then elements are swapped with new elements from a powerlaw distribution until the sequence makes a tree (#edges=#nodes-1). **Parameters**

`n:` the number of nodes

`gamma:` exponent of power law is gamma

`tries:` number of attempts to adjust sequence to make a tree

`seed:` seed for random number generator (default=None)

random_powerlaw_tree_sequence(*n*, *gamma*=3, *seed*=None, *tries*=100)

Return a degree sequence for a tree with a powerlaw distribution.

A trial powerlaw degree sequence is chosen and then elements are swapped with new elements from a powerlaw distribution until the sequence makes a tree (#edges=#nodes-1). **Parameters**

`n:` the number of nodes

`gamma:` exponent of power law is gamma

`tries:` number of attempts to adjust sequence to make a tree

`seed:` seed for random number generator (default=None)

26.2 Variables

Name	Description
__credits__	**Value:** `''`
__revision__	**Value:** `'$Revision: 1049 $'`
__package__	**Value:** `'networkx.generators'`

27 Module networkx.generators.small

Various small and named graphs, together with some compact generators. **Date:** $Date: 2005-06-15 12:53:08 -0600 (Wed, 15 Jun 2005) $

Author: Aric Hagberg (hagberg@lanl.gov) Pieter Swart (swart@lanl.gov)

27.1 Functions

make_small_graph(*graph_description*, *create_using*=None)

Return the small graph described by graph_description.

graph_description is a list of the form [ltype,name,n,xlist]

Here ltype is one of "adjacencylist" or "edgelist", name is the name of the graph and n the number of nodes. This constructs a graph of n nodes with integer labels 1,..,n.

If ltype="adjacencylist" then xlist is an adjacency list with exactly n entries, in with the j'th entry (which can be empty) specifies the nodes connected to vertex j. e.g. the "square" graph C_4 can be obtained by

```
>>> G=make_small_graph(["adjacencylist","C_4",4,[[2,4],[1,3],[2,4],[1,3]]])
```

or, since we do not need to add edges twice,

```
>>> G=make_small_graph(["adjacencylist","C_4",4,[[2,4],[3],[4],[]]])
```

If ltype="edgelist" then xlist is an edge list written as [[v1,w2],[v2,w2],...,[vk,wk]], where vj and wj integers in the range 1,..,n e.g. the "square" graph C_4 can be obtained by

```
>>> G=make_small_graph(["edgelist","C_4",4,[[1,2],[3,4],[2,3],[4,1]]])
```

Use the create_using argument to choose the graph class/type.

LCF_graph(*n*, *shift_list*, *repeats*)

Return the cubic graph specified in LCF notation.

LCF notation (LCF=Lederberg-Coxeter-Fruchte) is a compressed notation used in the generation of various cubic Hamiltonian graphs of high symmetry. See, for example, dodecahedral_graph, desargues_graph, heawood_graph and pappus_graph below.

n (number of nodes) The starting graph is the n-cycle with nodes 0,...,n-1. (The null graph is returned if n < 0.)

shift_list = [s1,s2,..,sk], a list of integer shifts mod n,

repeats integer specifying the number of times that shifts in shift_list are successively applied to each v_current in the n-cycle to generate an edge between v_current and v_current+shift mod n.

For v1 cycling through the n-cycle a total of k*repeats with shift cycling through shiftlist repeats times connect v1 with v1+shift mod n

The utility graph K_{3,3}

```
>>> G=LCF_graph(6,[3,-3],3)
```

The Heawood graph

```
>>> G=LCF_graph(14,[5,-5],7)
```

See http://mathworld.wolfram.com/LCFNotation.html for a description and references.

bull_graph()

Return the Bull graph.

chvatal_graph()

Return the Chvatal graph.

cubical_graph()

Return the 3-regular Platonic Cubical graph.

desargues_graph()

Return the Desargues graph.

diamond_graph()

Return the Diamond graph.

dodecahedral_graph()

Return the Platonic Dodecahedral graph.

frucht_graph()

Return the Frucht Graph.

The Frucht Graph is the smallest cubical graph whose automorphism group consists only of the identity element.

heawood_graph()

Return the Heawood graph, a (3,6) cage.

house_graph()

Return the House graph (square with triangle on top).

house_x_graph()

Return the House graph with a cross inside the house square.

icosahedral_graph()

Return the Platonic Icosahedral graph.

krackhardt_kite_graph()

Return the Krackhardt Kite Social Network.

A 10 actor social network introduced by David Krackhardt to illustrate: degree, betweenness, centrality, closeness, etc. The traditional labeling is: Andre=1, Beverley=2, Carol=3, Diane=4, Ed=5, Fernando=6, Garth=7, Heather=8, Ike=9, Jane=10.

moebius_kantor_graph()

Return the Moebius-Kantor graph.

octahedral_graph()

Return the Platonic Octahedral graph.

pappus_graph()

Return the Pappus graph.

petersen_graph()

Return the Petersen graph.

sedgewick_maze_graph()

Return a small maze with a cycle.

This is the maze used in Sedgewick,3rd Edition, Part 5, Graph Algorithms, Chapter 18, e.g. Figure 18.2 and following. Nodes are numbered 0,..,7

tetrahedral_graph()

Return the 3-regular Platonic Tetrahedral graph.

truncated_cube_graph()

Return the skeleton of the truncated cube.

truncated_tetrahedron_graph()

Return the skeleton of the truncated Platonic tetrahedron.

tutte_graph()

Return the Tutte graph.

27.2 Variables

Name	Description
__credits__	**Value:** `''`
__revision__	**Value:** `'$Revision: 1040 $'`
__package__	**Value:** `'networkx.generators'`

28 Module networkx.graph

Base class for graphs.

Examples Create an empty graph structure (a "null graph") with zero nodes and zero edges.

```
>>> from networkx import *
>>> G=Graph()
```

G can be grown in several ways. By adding one node at a time:

```
>>> G.add_node(1)
```

by adding a list of nodes:

```
>>> G.add_nodes_from([2,3])
```

by using an iterator:

```
>>> G.add_nodes_from(xrange(100,110))
```

or by adding any container of nodes

```
>>> H=path_graph(10)
>>> G.add_nodes_from(H)
```

H can be another graph, or dict, or set, or even a file. Any hashable object (except None) can represent a node, e.g. a Graph, a customized node object, etc.

```
>>> G.add_node(H)
```

G can also be grown by adding one edge at a time:

```
>>> G.add_edge( (1,2) )
```

by adding a list of edges:

```
>>> G.add_edges_from([(1,2),(1,3)])
```

or by adding any ebunch of edges (see above definition of an ebunch):

```
>>> G.add_edges_from(H.edges())
```

There are no complaints when adding existing nodes or edges:

```
>>> G=Graph()
>>> G.add_edge([(1,2),(1,3)])
```

will add new nodes as required. **Author:** Aric Hagberg (hagberg@lanl.gov) Pieter Swart (swart@lanl.gov) Dan Schult(dschult@colgate.edu)

28.1 Variables

Name	Description
__package__	**Value:** `'networkx'`

28.2 Class Graph

object
└
networkx.graph.Graph

Known Subclasses: networkx.digraph.DiGraph, networkx.xgraph.XGraph, networkx.tree.Tree, networkx.tree.Forest, networkx.tree.RootedTree

Graph is a simple graph without any multiple (parallel) edges or self-loops. Attempting to add either will not change the graph and will not report an error.

28.2.1 Methods

__init__(*self*, *data*=None, *name*='')

Initialize Graph.

```
>>> G=Graph(name="empty")
```

creates empty graph G with G.name="empty" Overrides: object.__init__

__str__(*self*)

str(x) Overrides: object.__str__ extit(inherited documentation)

__iter__(*self*)

Return an iterator over the nodes in G.

This is the iterator for the underlying adjacency dict. (Allows the expression 'for n in G')

__contains__(*self*, *n*)

Return True if n is a node in graph.

Allows the expression 'n in G'.

Testing whether an unhashable object, such as a list, is in the dict datastructure (self.adj) will raise a TypeError. Rather than propagate this to the calling method, just return False.

__len__(*self*)

Return the number of nodes in graph.

__getitem__(*self*, *n*)

Return the neighbors of node n as a list.

This provides graph G the natural property that G[n] returns the neighbors of G.

prepare_nbunch(*self*, *nbunch*=`None`)

Return a sequence (or iterator) of nodes contained in nbunch which are also in the graph.

The input nbunch can be a single node, a sequence or iterator of nodes or None (omitted). If None, all nodes in the graph are returned.

Note: This routine exhausts any iterator nbunch.

Note: To test whether nbunch is a single node, one can use "if nbunch in self:", even after processing with this routine.

Note: This routine returns an empty list if nbunch is not either a node, sequence, iterator, or None. You can catch this exception if you want to change this behavior.

info(*self*, *n*=None)

Print short info for graph G or node n.

add_node(*self*, *n*)

Add a single node n to the graph.

The node n can be any hashable object except None.

A hashable object is one that can be used as a key in a Python dictionary. This includes strings, numbers, tuples of strings and numbers, etc. On many platforms this also includes mutables such as Graphs e.g., though one should be careful the hash doesn't change on mutables.

Example:

```
>>> from networkx import *
>>> G=Graph()
>>> K3=complete_graph(3)
>>> G.add_node(1)
>>> G.add_node('Hello')
>>> G.add_node(K3)
>>> G.number_of_nodes()
3
```

add_nodes_from(*self*, *nlist*)

Add multiple nodes to the graph.

nlist: A container of nodes that will be iterated through once (thus it should be an iterator or be iterable). Each element of the container should be a valid node type: any hashable type except None. See add_node for details.

Examples:

```
>>> from networkx import *
>>> G=Graph()
>>> K3=complete_graph(3)
>>> G.add_nodes_from('Hello')
>>> G.add_nodes_from(K3)
>>> sorted(G.nodes())
[0, 1, 2, 'H', 'e', 'l', 'o']
```

delete_node(*self*, *n*)

Delete node n from graph. Attempting to delete a non-existent node will raise an exception.

delete_nodes_from(*self*, *nlist*)

Remove nodes in nlist from graph.

nlist: an iterable or iterator containing valid node names.

Attempting to delete a non-existent node will raise an exception. This could mean some nodes got deleted and other valid nodes did not.

nodes_iter(*self*)

Return an iterator over the graph nodes.

nodes(*self*)

Return a copy of the graph nodes in a list.

number_of_nodes(*self*)

Return number of nodes.

has_node(*self*, *n*)

Return True if graph has node n.

(duplicates self.__contains__) "n in G" is a more readable version of "G.has_node(n)"?

order(*self*)

Return the order of a graph = number of nodes.

add_edge(*self*, *u*, *v*=None)

Add a single edge (u,v) to the graph.

>> G.add_edge(u,v) and >>> G.add_edge((u,v)) are equivalent forms of adding a single edge between nodes u and v. The nodes u and v will be automatically added if not already in the graph. They must be a hashable (except None) Python object.

The following examples all add the edge (1,2) to graph G.

```
>>> G=Graph()
>>> G.add_edge( 1, 2 )           # explicit two node form
>>> G.add_edge( (1,2) )          # single edge as tuple of two nodes
>>> G.add_edges_from( [(1,2)] ) # add edges from iterable container
```

add_edges_from(*self*, *ebunch*)

Add all the edges in ebunch to the graph.

ebunch: Container of 2-tuples (u,v). The container must be iterable or an iterator. It is iterated over once. Adding the same edge twice has no effect and does not raise an exception.

delete_edge(*self*, *u*, *v*=None)

Delete the single edge (u,v).

Can be used in two basic forms: >>> G.delete_edge(u,v) and >> G.delete_edge((u,v)) are equivalent ways of deleting a single edge between nodes u and v.

Return without complaining if the nodes or the edge do not exist.

delete_edges_from(*self*, *ebunch*)

Delete the edges in ebunch from the graph.

ebunch: an iterator or iterable of 2-tuples (u,v).

Edges that are not in the graph are ignored.

has_edge(*self*, *u*, *v*=`None`)

Return True if graph contains the edge u-v, return False otherwise.

has_neighbor(*self*, *u*, *v*)

Return True if node u has neighbor v.

This is equivalent to has_edge(u,v).

get_edge(*self*, *u*, *v*=`None`)

Return 1 if graph contains the edge u-v. Raise an exception otherwise.

neighbors_iter(*self*, *n*)

Return an iterator over all neighbors of node n.

neighbors(*self*, *n*)

Return a list of nodes connected to node n.

edges_iter(*self*, *nbunch*=`None`)

Return iterator that iterates once over each edge adjacent to nodes in nbunch, or over all edges in graph if no nodes are specified.

If nbunch is None return all edges in the graph. The argument nbunch can be any single node, or any sequence or iterator of nodes. Nodes in nbunch that are not in the graph will be (quietly) ignored.

edges(*self*, *nbunch*=`None`)

Return list of all edges that are adjacent to a node in nbunch, or a list of all edges in graph if no nodes are specified.

If nbunch is None return all edges in the graph. The argument nbunch can be any single node, or any sequence or iterator of nodes. Nodes in nbunch that are not in the graph will be (quietly) ignored.

For digraphs, edges=out_edges

edge_boundary(*self*, *nbunch1*, *nbunch2*=`None`)

Return list of edges (n1,n2) with n1 in nbunch1 and n2 in nbunch2. If nbunch2 is omitted or nbunch2=None, then nbunch2 is all nodes not in nbunch1.

Nodes in nbunch1 and nbunch2 that are not in the graph are ignored.

nbunch1 and nbunch2 are usually meant to be disjoint, but in the interest of speed and generality, that is not required here.

This routine is faster if nbunch1 is smaller than nbunch2.

node_boundary(*self*, *nbunch1*, *nbunch2*=`None`)

Return list of all nodes on external boundary of nbunch1 that are in nbunch2. If nbunch2 is omitted or nbunch2=None, then nbunch2 is all nodes not in nbunch1.

Note that by definition the node_boundary is external to nbunch1.

Nodes in nbunch1 and nbunch2 that are not in the graph are ignored.

nbunch1 and nbunch2 are usually meant to be disjoint, but in the interest of speed and generality, that is not required here.

This routine is faster if nbunch1 is smaller than nbunch2.

degree(*self*, *nbunch*=`None`, *with_labels*=`False`)

Return degree of single node or of nbunch of nodes. If nbunch is omitted or nbunch=None, then return degrees of *all* nodes.

The degree of a node is the number of edges attached to that node.

Can be called in three ways:

G.degree(n): return the degree of node n G.degree(nbunch): return a list of values, one for each n in nbunch (nbunch is any iterable container of nodes.) G.degree(): same as nbunch = all nodes in graph.

If with_labels==True, then return a dict that maps each n in nbunch to degree(n).

Any nodes in nbunch that are not in the graph are (quietly) ignored.

degree_iter(*self*, *nbunch*=`None`, *with_labels*=`False`)

Return iterator that return degree(n) or (n,degree(n)) for all n in nbunch. If nbunch is ommitted, then iterate over all nodes.

Can be called in three ways: G.degree_iter(n): return iterator the degree of node n G.degree_iter(nbunch): return a list of values, one for each n in nbunch (nbunch is any iterable container of nodes.) G.degree_iter(): same as nbunch = all nodes in graph.

If with_labels==True, iterator will return an (n,degree(n)) tuple of node and degree.

Any nodes in nbunch that are not in the graph are (quietly) ignored.

clear(*self*)

Remove name and delete all nodes and edges from graph.

copy(*self*)

Return a (shallow) copy of the graph.

Identical to dict.copy() of adjacency dict adj, with name copied as well.

to_undirected(*self*)

Return the undirected representation of the graph G.

This graph is undirected, so merely return a copy.

to_directed(*self*)

Return a directed representation of the graph G.

A new digraph is returned with the same name, same nodes and with each edge u-v represented by two directed edges u->v and v->u.

subgraph(*self*, *nbunch*, *inplace*=`False`, *create_using*=`None`)

Return the subgraph induced on nodes in nbunch.

nbunch: can be a single node or any iterable container of of nodes. (It can be an iterable or an iterator, e.g. a list, set, graph, file, numeric array, etc.)

Setting inplace=True will return the induced subgraph in the original graph by deleting nodes not in nbunch. This overrides create_using. Warning: this can destroy the graph.

Unless otherwise specified, return a new graph of the same type as self. Use (optional) create_using=R to return the resulting subgraph in R. R can be an existing graph-like object (to be emptied) or R can be a call to a graph object, e.g. create_using=DiGraph(). See documentation for empty_graph()

Note: use subgraph(G) rather than G.subgraph() to access the more general subgraph() function from the operators module.

add_path(*self*, *nlist*)

Add the path through the nodes in nlist to graph

add_cycle(*self*, *nlist*)

Add the cycle of nodes in nlist to graph

is_directed(*self*)

Return True if graph is directed.

size(*self*)

Return the size of a graph = number of edges.

number_of_edges(*self*, *u*=None, *v*=None)

Return the number of edges between nodes u and v.

If u and v are not specified return the number of edges in the entire graph.

The edge argument e=(u,v) can be specified as G.number_of_edges(u,v) or G.number_of_edges(e)

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __new__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __subclasshook__()

28.2.2 Properties

Name	Description
Inherited from object	
__class__	

29 Module networkx.hybrid

Hybrid **Date:** $Date: 2005-03-30 16:56:28 -0700 (Wed, 30 Mar 2005) $

Author: Aric Hagberg (hagberg@lanl.gov) Dan Schult (dschult@colgate.edu)

29.1 Functions

kl_connected_subgraph(*G*, *k*, *l*, *low_memory*=`False`, *same_as_graph*=`False`)

Returns the maximum locally (k,l) connected subgraph of G.

(k,l)-connected subgraphs are presented by Fan Chung and Li in "The Small World Phenomenon in hybrid power law graphs" to appear in "Complex Networks" (Ed. E. Ben-Naim) Lecture Notes in Physics, Springer (2004)

low_memory=True then use a slightly slower, but lower memory version same_as_graph=True then return a tuple with subgraph and pflag for if G is kl-connected

is_kl_connected(*G*, *k*, *l*, *low_memory*=`False`)

Returns True if G is kl connected

29.2 Variables

Name	Description
__credits__	**Value:** `''`
__revision__	**Value:** `'$Revision: 911 $'`
__package__	**Value:** `'networkx'`
__warningregistry__	**Value:** `{('the sets module is deprecated', <type 'exceptions.Depr...`

30 Module networkx.info

Graph classes Graph

A simple graph that has no self-loops or multiple (parallel) edges.

An empty graph is created with

```
>>> G=Graph()
```

DiGraph

A directed graph that has no self-loops or multiple (parallel) edges. Subclass of Graph.

An empty digraph is created with

```
>>> G=DiGraph()
```

XGraph

A graph that has (optional) self-loops or multiple (parallel) edges and arbitrary data on the edges. Subclass of Graph.

An empty graph is created with

```
>>> G=XGraph()
```

XDiGraph

A directed graph that has (optional) self-loops or multiple (parallel) edges and arbitrary data on the edges.

A simple digraph that has no self-loops or multiple (parallel) edges. Subclass of DiGraph which is a subclass of Graph.

An empty digraph is created with

```
>>> G=DiGraph()
```

The XGraph and XDiGraph classes extend the Graph and DiGraph classes by allowing (optional) self loops, multiedges and by decorating each edge with an object x.

Each XDiGraph or XGraph edge is a 3-tuple e=(n1,n2,x), representing an edge between nodes n1 and n2 that is decorated with the object x. Here n1 and n2 are (hashable) node objects and x is a (not necessarily hashable) edge object. If multiedges are allowed, G.get_edge(n1,n2) returns a list of edge objects.

Whether an XGraph or XDiGraph allow self-loops or multiple edges is determined initially via parameters selfloops=True/False and multiedges=True/False. For example, the example empty XGraph created above is equivalent to

```
>>> G=XGraph(selfloops=False, multiedges=False)
```

Similar defaults hold for XDiGraph. The command

```
>>> G=XDiGraph(multiedges=True)
```

creates an empty digraph G that does not allow selfloops but does allow for multiple (parallel) edges. Methods exist for allowing or disallowing each feature after instatiation as well.

Note that if G is an XGraph then G.add_edge(n1,n2) will add the edge (n1,n2,None), and G.delete_edge(n1,n2) will attempt to delete the edge (n1,n2,None). In the case of multiple edges between nodes n1 and n2, one can use G.delete_multiedge(n1,n2) to delete all edges between n1 and n2.

Notation The following shorthand is used throughout NetworkX documentation and code: (we use mathematical notation n,v,w,... to indicate a node, v=vertex=node).

G,G1,G2,H,etc: Graphs

n,n1,n2,u,v,v1,v2: nodes (vertices)

nlist: a list of nodes (vertices)

nbunch: a "bunch" of nodes (vertices). An nbunch is either a single node of the graph or any iterable container/iterator of nodes. The distinction is determined by checking if nbunch is in the graph. If you use iterable containers as nodes you should be careful when using nbunch.

e=(n1,n2): an edge (a python "2-tuple"), also written n1-n2 (if undirected) and n1->n2 (if directed).

e=(n1,n2,x): an edge triple ("3-tuple") containing the two nodes connected and the edge data/label/object stored associated with the edge. The object x, or a list of objects (if multiedges=True), can be obtained using G.get_edge(n1,n2)

elist: a list of edges (as 2- or 3-tuples)

ebunch: a bunch of edges (as 2- or 3-tuples). An ebunch is any iterable (non-string) container of edge-tuples (either 2-tuples, 3-tuples or a mixture).

Warning:

- The ordering of objects within an arbitrary nbunch/ebunch can be machine-

dependent.

- Algorithms should treat an arbitrary nbunch/ebunch as once-through-and-exhausted iterable containers.
- len(nbunch) and len(ebunch) need not be defined.

Methods Each class provides basic graph methods.

Mutating Graph methods

- G.add_node(n), G.add_nodes_from(nlist)
- G.delete_node(n), G.delete_nodes_from(nlist)
- G.add_edge(n1,n2), G.add_edge(e), where e=(u,v)
- G.add_edges_from(ebunch)
- G.delete_edge(n1,n2), G.delete_edge(e), where e=(u,v)
- G.delete_edges_from(ebunch)
- G.add_path(nlist)
- G.add_cycle(nlist)
- G.clear()
- G.subgraph(nbunch,inplace=True)

Non-mutating Graph methods

- len(G)
- G.has_node(n)
- n in G (equivalent to G.has_node(n))
- for n in G: (iterate through the nodes of G)
- G.nodes()
- G.nodes_iter()

- G.has_edge(n1,n2), G.has_neighbor(n1,n2), G.get_edge(n1,n2)
- G.edges(), G.edges(n), G.edges(nbunch)
- G.edges_iter(), G.edges_iter(n), G.edges_iter(nbunch)
- G.neighbors(n)
- G[n] (equivalent to G.neighbors(n))
- G.neighbors_iter(n) # iterator over neighbors
- G.number_of_nodes(), G.order()
- G.number_of_edges(), G.size()
- G.edge_boundary(nbunch1), G.node_boundary(nbunch1)
- G.degree(n), G.degree(nbunch)
- G.degree_iter(n), G.degree_iter(nbunch)
- G.is_directed()
- G.info() # print variaous info about a graph
- G.prepare_nbunch(nbunch) # return list of nodes in G and nbunch

Methods returning a new graph

- G.subgraph(nbunch)
- G.subgraph(nbunch,create_using=H)
- G.copy()
- G.to_undirected()
- G.to_directed()

Implementation Notes The graph classes implement graphs using data structures based on an adjacency list implemented as a node-centric dictionary of dictionaries. The dictionary contains keys corresponding to the nodes and the values are dictionaries of neighboring node keys with the value None (the Python None type) for Graph and DiGraph or user specified (default is None) for XGraph and XDiGraph. The dictionary of dictionary structure allows fast addition, deletion and lookup of nodes and neighbors in large graphs.

Similarities between XGraph and Graph XGraph and Graph differ in the way edge data is handled. XGraph edges are 3-tuples (n1,n2,x) and Graph edges are 2-tuples (n1,n2). XGraph inherits from the Graph class, and XDiGraph from the DiGraph class.

Graph and XGraph are similar in the following ways:

1. Edgeless graphs are the same in XGraph and Graph. For an edgeless graph, represented by G (member of the Graph class) and XG (member of XGraph class), there is no difference between the datastructures G.adj and XG.adj, other than possibly in the ordering of the keys in the adj dict.

2. Basic graph construction code for G=Graph() will also work for G=XGraph(). In the Graph class, the simplest graph construction consists of a graph creation command G=Graph() followed by a list of graph construction commands, consisting of successive calls to the methods:

   G.add_node, G.add_nodes_from, G.add_edge, G.add_edges, G.add_path, G.add_cycle
   G.delete_node, G.delete_nodes_from, G.delete_edge, G.delete_edges_from

   with all edges specified as 2-tuples,

   If one replaces the graph creation command with G=XGraph(), and then apply the identical list of construction commands, the resulting XGraph object will be a simple graph G with identical datastructure G.adj. This property ensures reuse of code developed for graph generation in the Graph class.

30.1 Variables

Name	Description
__package__	**Value:** `None`

31 Module networkx.isomorph

Fast checking to see if graphs are not isomorphic.

This isn't a graph isomorphism checker. **Date:** $Date: 2005-05-31 17:00:13 -0600 (Tue, 31 May 2005) $

Author: Pieter Swart (swart@lanl.gov) Dan Schult (dschult@colgate.edu)

31.1 Functions

graph_could_be_isomorphic(*G1*, *G2*)

Returns False if graphs G1 and G2 are definitely not isomorphic. True does NOT garantee isomorphism.

Checks for matching degree, triangle, and number of cliques sequences.

fast_graph_could_be_isomorphic(*G1*, *G2*)

Returns False if graphs G1 and G2 are definitely not isomorphic. True does NOT garantee isomorphism.

Checks for matching degree and triangle sequences.

faster_graph_could_be_isomorphic(*G1*, *G2*)

Returns False if graphs G1 and G2 are definitely not isomorphic. True does NOT garantee isomorphism.

Checks for matching degree sequences in G1 and G2.

is_isomorphic(*G1*, *G2*)

Returns True if the graphs G1 and G2 are isomorphic and False otherwise.

Uses the vf2 algorithm - see networkx.isomorphvf2

31.2 Variables

Name	Description
__credits__	**Value:** `''`
__revision__	**Value:** `'$Revision: 1002 $'`
__package__	**Value:** `'networkx'`

32 Module networkx.isomorphvf2

An implementation of VF2 algorithm for graph ismorphism testing, as seen here:

> **Luigi P. Cordella, Pasquale Foggia, Carlo Sansone, Mario Vento,** "A (Sub)Graph Isomorphism Algorithm for Matching Large Graphs," IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 26, no. 10, pp. 1367-1372, Oct., 2004.

Modified to handle undirected graphs. Modified to handle multiple edges. **Date:** $Date: 2007-08-21 16:49:09 -0600 (Tue, 21 Aug 2007) $

32.1 Variables

Name	Description
__credits__	**Value:** `'$Credits:$'`
__revision__	**Value:** `'$Revision:  680 $'`
__package__	**Value:** `'networkx'`

32.2 Class GraphMatcher

object
└ **networkx.isomorphvf2.GraphMatcher**

A GraphMatcher is responsible for matching undirected graphs (Graph or XGraph) in a predetermined manner. For graphs G1 and G2, this typically means a check for an isomorphism between them, though other checks are also possible. For example, the GraphMatcher class can check if a subgraph of G1 is isomorphic to G2.

Matching is done via syntactic feasibility. It is also possible to check for semantic feasibility. Feasibility, then, is defined as the logical AND of the two functions.

To include a semantic check, the GraphMatcher class should be subclassed, and the semantic_feasibility() function should be redefined. By default, the semantic feasibility function always returns True. The effect of this is that semantics are not considered in the matching of G1 and G2.

For more information, see the docmentation for: syntactic_feasibliity() semantic_feasibility()

Suppose G1 and G2 are isomorphic graphs. Verification is as follows:

```
>>> GM = GraphMatcher(G1,G2)
>>> GM.is_isomorphic()
```

```
True
>>> GM.mapping
```

GM.mapping stores the isomorphism mapping.

32.2.1 Methods

__init__(*self*, *G1*, *G2*)

Initialize GraphMatcher.

Suppose G1 and G2 are undirected graphs.

```
>>> GM = GraphMatcher(G1,G2)
```

creates a GraphMatcher which only checks for syntactic feasibility. Overrides: object.__init__

__del__(*self*)

candidate_pairs_iter(*self*)

This function returns an iterator over pairs to be considered for inclusion in the current partial isomorphism mapping.

is_isomorphic(*self*)

Returns True if G1 and G2 are isomorphic graphs. Otherwise, it returns False.

match(*self*, *state*)

This function is called recursively to determine if a complete isomorphism can be found between G1 and G2. It cleans up the class variables after each recursive call. If an isomorphism is found, we raise a StopIteration and jump immediately out of the recursion.

semantic_feasibility(*self*, *G1_node*, *G2_node*)

The semantic feasibility function should return True if it is acceptable to add the candidate pair (G1_node, G2_node) to the current partial isomorphism mapping. The logic should focus on semantic information contained in the edge data or a formalized node class.

By acceptable, we mean that the subsequent mapping can still become a complete isomorphism mapping. Thus, if adding the candidate pair definitely makes it so that the subsequent mapping cannot become a complete isomorphism mapping, then this function must return False.

The default semantic feasibility function always returns True. The effect is that semantics are not considered in the matching of G1 and G2.

The semantic checks might differ based on the what type of test is being performed. A keyword description of the test is stored in self.test. Here is a quick description of the currently implemented tests:

test='graph' Indicates that the graph matcher is looking for a graph-graph isomorphism.

test='subgraph' Indicates that the graph matcher is looking for a subgraph-graph isomorphism such that a subgraph of G1 is isomorphic to G2.

Any subclass of GraphMatcher which redefines semantic_feasibility() must maintain the above form to keep the match() method functional. Implementation considerations should include directed and undirected graphs, as well as graphs with multiple edges.

As an example, if edges have weights, one feasibility function would be to demand that the weight values/relationships are preserved in the isomorphism mapping.

subgraph_is_isomorphic(*self*)

Returns True if a subgraph of G1 is isomorphic to G2. Otherwise, it returns False.

syntactic_feasibility(*self*, *G1_node*, *G2_node*)

This function returns True if it is adding the candidate pair to the current partial isomorphism mapping is allowable. The addition is allowable if the inclusion of the candidate pair does not make it impossible for an isomorphism to be found.

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __new__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

32.2.2 Properties

Name	Description
Inherited from object	
__class__	

32.3 Class DiGraphMatcher

object
└
networkx.isomorphvf2.DiGraphMatcher

A DiGraphMatcher is responsible for matching directed graphs (DiGraph or XDiGraph) in a predetermined manner. For graphs G1 and G2, this typically means a check for an isomorphism between them, though other checks are also possible. For example, the DiGraphMatcher class can check if a subgraph of G1 is isomorphic to G2.

Matching is done via syntactic feasibility. It is also possible to check for semantic feasibility. Feasibility, then, is defined as the logical AND of the two functions.

To include a semantic check, you should subclass the GraphMatcher class and redefine semantic_feasibility(). By default, the semantic feasibility function always returns True. The effect of this is that semantics are not considered in the matching of G1 and G2.

For more information, see the docmentation for: syntactic_feasibliity() semantic_feasibility()

Suppose G1 and G2 are isomorphic graphs. Verfication is as follows:

```
>>> GM = GraphMatcher(G1,G2)
>>> GM.is_isomorphic()
True
>>> GM.mapping
```

GM.mapping stores the isomorphism mapping.

32.3.1 Methods

__init__(*self*, *G1*, *G2*)

Initialize DiGraphMatcher.

Suppose G1 and G2 are graphs.

```
>>> GM = DiGraphMatcher(G1,G2)
```

creates a DiGraphMatcher which only checks for syntactic feasibility.
Overrides: object.__init__

__del__(*self*)

candidate_pairs_iter(*self*)

This function returns an iterator over pairs to be considered for inclusion in the current partial isomorphism mapping.

is_isomorphic(*self*)

Returns True if G1 and G2 are isomorphic graphs. Otherwise, it returns False.

match(*self*, *state*)

This function is called recursively to determine if a complete isomorphism can be found between G1 and G2. It cleans up the class variables after each recursive call. Because of this, this function will not return True or False. If a mapping is found, we will jump out of the recursion by throwing an exception. Otherwise, we will return nothing.

semantic_feasibility(*self*, *G1_node*, *G2_node*)

The semantic feasibility function should return True if it is acceptable to add the candidate pair (G1_node, G2_node) to the current partial isomorphism mapping. The logic should focus on semantic information contained in the edge data or a formalized node class.

By acceptable, we mean that the subsequent mapping can still become a complete isomorphism mapping. Thus, if adding the candidate pair definitely makes it so that the subsequent mapping cannot become a complete isomorphism mapping, then this function must return False.

The default semantic feasibility function always returns True. The effect is that semantics are not considered in the matching of G1 and G2.

The semantic checks might differ based on the what type of test is being performed. A keyword description of the test is stored in self.test. Here is a quick description of the currently implemented tests:

test='graph' Indicates that the graph matcher is looking for a graph-graph isomorphism.

test='subgraph' Indicates that the graph matcher is looking for a subgraph-graph isomorphism such that a subgraph of G1 is isomorphic to G2.

Any subclass of DiGraphMatcher which redefines semantic_feasibility() must maintain the above form to keep the match() method functional. Implementation considerations should include directed and undirected graphs, as well as graphs with multiple edges.

As an example, if edges have weights, one feasibility function would be to demand that the weight values/relationships are preserved in the isomorphism mapping.

subgraph_is_isomorphic(*self*)

Returns True if a subgraph of G1 is isomorphic to G2. Otherwise, it returns False.

syntactic_feasibility(*self*, *G1_node*, *G2_node*)

This function returns True if it is adding the candidate pair to the current partial isomorphism mapping is allowable.

Keywords:

test='graph' Checks for graph-graph isomorphism. This is the default value.

test='subgraph' Checks for graph-subgraph isomorphism in such a way that a subgraph of G1 might be isomorphic to G2.

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __new__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

32.3.2 Properties

Name	Description
Inherited from object	
__class__	

32.4 Class GMState

object
└ **networkx.isomorphvf2.GMState**

This class is used internally by the GraphMatcher class. It is used only to store state specific data. There will be at most G2.order() of these objects in memory at a time, due to the depth-first search strategy employed by the VF2 algorithm.

32.4.1 Methods

__init__(*self*, *GM*, *G1_node*=None, *G2_node*=None)

Initializes GMState object.

Pass in the GraphMatcher to which this DiGMState belongs and the new node pair that will be added to the GraphMatcher's current isomorphism mapping.
Overrides: object.__init__

__del__(*self*)

Deletes the GMState object and restores the class variables.

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __new__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

32.4.2 Properties

Name	Description
Inherited from object	
__class__	

32.4.3 Class Variables

Name	Description
core_1	**Value:** {}
core_2	**Value:** {}
inout_1	**Value:** {}
inout_2	**Value:** {}

32.5 Class DiGMState

object
└ **networkx.isomorphvf2.DiGMState**

This class is used internally by the DiGraphMatcher class. It is used only to store state specific data. There will be at most G2.order() of these objects in memory at a time, due to the depth-first search strategy employed by the VF2 algorithm.

32.5.1 Methods

__init__(*self*, *DiGM*, *G1_node*=None, *G2_node*=None)

Initializes DiGMState object.

Pass in the DiGraphMatcher to which this DiGMState belongs and the new node pair that will be added to the GraphMatcher's current isomorphism mapping. Overrides: object.__init__

__del__(*self*)

Deletes the DiGMState object and restores the class variables.

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __new__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

32.5.2 Properties

Name	Description
Inherited from object	
__class__	

32.5.3 Class Variables

Name	Description
core_1	**Value:** {}
core_2	**Value:** {}
in_1	**Value:** {}
in_2	**Value:** {}
out_1	**Value:** {}
out_2	**Value:** {}

33 Module networkx.operators

Operations on graphs; including union, complement, subgraph. **Date:** $Date: 2007-07-18 15:23:23 -0600 (Wed, 18 Jul 2007) $

Author: Aric Hagberg (hagberg@lanl.gov) Pieter Swart (swart@lanl.gov) Dan Schult(dschult@colgate.edu

33.1 Functions

subgraph(*G*, *nbunch*, *inplace*=`False`, *create_using*=`None`)

Return the subgraph induced on nodes in nbunch.

nbunch: can be a singleton node, a string (which is treated as a singleton node), or any iterable container of of nodes. (It can be an iterable or an iterator, e.g. a list, set, graph, file, numeric array, etc.)

Setting inplace=True will return the induced subgraph in the original graph by deleting nodes not in nbunch. This overrides create_using. Warning: this can destroy the graph.

Unless otherwise specified, return a new graph of the same type as self. Use (optional) create_using=R to return the resulting subgraph in R. R can be an existing graph-like object (to be emptied) or R is a call to a graph object, e.g. create_using=DiGraph(). See documentation for empty_graph.

Implemented for Graph, DiGraph, XGraph, XDiGraph

Note: subgraph(G) calls G.subgraph()

union(*G*, *H*, *create_using*=`None`, *rename*=`False`, *name*=`None`)

Return the union of graphs G and H.

Graphs G and H must be disjoint, otherwise an exception is raised.

Node names of G and H can be changed be specifying the tuple rename=('G-','H-') (for example). Node u in G is then renamed "G-u" and v in H is renamed "H-v".

To force a disjoint union with node relabeling, use disjoint_union(G,H) or convert_node_labels_to integers().

Optional create_using=R returns graph R filled in with the union of G and H. Otherwise a new graph is created, of the same class as G. It is recommended that G and H be either both directed or both undirected.

A new name can be specified in the form X=graph_union(G,H,name="new_name")

Implemented for Graph, DiGraph, XGraph, XDiGraph.

disjoint_union(*G*, *H*)

Return the disjoint union of graphs G and H, forcing distinct integer node labels.

A new graph is created, of the same class as G. It is recommended that G and H be either both directed or both undirected.

Implemented for Graph, DiGraph, XGraph, XDiGraph.

cartesian_product(*G*, *H*)

Return the Cartesian product of G and H.

Tested only on Graph class.

compose(*G*, *H*, *create_using*=`None`, *name*=`None`)

Return a new graph of G composed with H.

The node sets of G and H need not be disjoint.

A new graph is returned, of the same class as G. It is recommended that G and H be either both directed or both undirected.

Optional create_using=R returns graph R filled in with the compose(G,H). Otherwise a new graph is created, of the same class as G. It is recommended that G and H be either both directed or both undirected.

Implemented for Graph, DiGraph, XGraph, XDiGraph

complement(*G*, *create_using*=`None`, *name*=`None`)

Return graph complement of G.

Unless otherwise specified, return a new graph of the same type as self. Use (optional) create_using=R to return the resulting subgraph in R. R can be an existing graph-like object (to be emptied) or R can be a call to a graph object, e.g. create_using=DiGraph(). See documentation for empty_graph()

Implemented for Graph, DiGraph, XGraph, XDiGraph. Note that complement() is not well-defined for XGraph and XDiGraph objects that allow multiple edges or self-loops.

create_empty_copy(*G*, *with_nodes*=`True`)

Return a copy of the graph G with all of the edges removed.

convert_to_undirected(*G*)

Return a new undirected representation of the graph G.

Works for Graph, DiGraph, XGraph, XDiGraph.

Note: convert_to_undirected(G)=G.to_undirected()

convert_to_directed(*G*)

Return a new directed representation of the graph G.

Works for Graph, DiGraph, XGraph, XDiGraph.

Note: convert_to_directed(G)=G.to_directed()

relabel_nodes(*G*, *mapping*)

Return a copy of G with node labels transformed by mapping.

mapping is either

- a dictionary with the old labels as keys and new labels as values
- a function transforming an old label with a new label

In either case, the new labels must be hashable Python objects.

mapping as dictionary:

```
>>> G=path_graph(3)  # nodes 0-1-2
>>> mapping={0:'a',1:'b',2:'c'}
>>> H=relabel_nodes(G,mapping)
>>> print H.nodes()
['a', 'c', 'b']

>>> G=path_graph(26) # nodes 0..25
>>> mapping=dict(zip(G.nodes(),"abcdefghijklmnopqrstuvwxyz"))
>>> H=relabel_nodes(G,mapping) # nodes a..z
>>> mapping=dict(zip(G.nodes(),xrange(1,27)))
>>> G1=relabel_nodes(G,mapping) # nodes 1..26
```

mapping as function

```
>>> G=path_graph(3)
>>> def mapping(x):
...     return x**2
>>> H=relabel_nodes(G,mapping)
>>> print H.nodes()
[0, 1, 4]
```

Also see convert_node_labels_to_integers.

relabel_nodes_with_function(*G*, *func*)

Deprecated: call relabel_nodes(G,func).

convert_node_labels_to_integers(*G*, *first_label*=`0`, *ordering*=`'default'`, *discard_old_labels*=`True`)

Return a copy of G, with n node labels replaced with integers, starting at first_label.

first_label: (optional, default=0)

An integer specifying the offset in numbering nodes. The n new integer labels are numbered first_label, ..., n+first_label.

ordering: (optional, default="default")

A string specifying how new node labels are ordered. Possible values are:

"default" : inherit node ordering from G.nodes() "sorted" : inherit node ordering from sorted(G.nodes()) "increasing degree" : nodes are sorted by increasing degree "decreasing degree" : nodes are sorted by decreasing degree

discard_old_labels if True (default) discard old labels if False, create a dict self.node_labels that maps new labels to old labels

Works for Graph, DiGraph, XGraph, XDiGraph

33.2 Variables

Name	Description
__credits__	**Value:** `''`
__revision__	**Value:** `'$Revision: 1024 $'`
__package__	**Value:** `'networkx'`

34 Module networkx.path

Shortest path algorithms. **Author:** Aric Hagberg (hagberg@lanl.gov)

34.1 Functions

shortest_path_length(*G*, *source*, *target*)

Return the shortest path length in the graph G between the source and target. Raise an exception if no path exists.

G is treated as an unweighted graph. For weighted graphs see dijkstra_path_length.

single_source_shortest_path_length(*G*, *source*, *cutoff*=None)

Shortest path length from source to all reachable nodes.

Returns a dictionary of shortest path lengths keyed by target.

```
>>> G=path_graph(5)
>>> length=single_source_shortest_path_length(G,1)
>>> length[4]
3
>>> print length
{0: 1, 1: 0, 2: 1, 3: 2, 4: 3}
```

cutoff is optional integer depth to stop the search - only paths of length <= cutoff are returned.

all_pairs_shortest_path_length(*G*, *cutoff*=None)

Return dictionary of shortest path lengths between all nodes in G.

The dictionary only has keys for reachable node pairs. >>> G=path_graph(5) >>> length=all_pairs_shortest_path_length(G) >>> print length[1][4] 3 >>> length[1] {0: 1, 1: 0, 2: 1, 3: 2, 4: 3}

cutoff is optional integer depth to stop the search - only paths of length <= cutoff are returned.

shortest_path(*G*, *source*, *target*)

Return a list of nodes in G for a shortest path between source and target.

There may be more than one shortest path. This returns only one.

bidirectional_shortest_path(*G*, *source*, *target*)

Return list of nodes in a shortest path between source and target. Return False if no path exists.

Also known as shortest_path.

single_source_shortest_path(*G*, *source*, *cutoff*=`None`)

Return list of nodes in a shortest path between source and all other nodes in G reachable from source.

There may be more than one shortest path between the source and target nodes - this routine returns only one.

cutoff is optional integer depth to stop the search - only paths of length <= cutoff are returned.

See also shortest_path and bidirectional_shortest_path.

all_pairs_shortest_path(*G*, *cutoff*=`None`)

Return dictionary of shortest paths between all nodes in G.

The dictionary only has keys for reachable node pairs.

cutoff is optional integer depth to stop the search - only paths of length <= cutoff are returned.

See also floyd_warshall.

dijkstra_path(*G*, *source*, *target*)

Returns the shortest path from source to target in a weighted graph G. Uses a bidirectional version of Dijkstra's algorithm.

Edge data must be numerical values for XGraph and XDiGraphs. The weights are assigned to be 1 for Graphs and DiGraphs.

See also bidirectional_dijkstra for more information about the algorithm.

dijkstra_path_length(*G*, *source*, *target*)

Returns the shortest path length from source to target in a weighted graph G. Uses a bidirectional version of Dijkstra's algorithm.

Edge data must be numerical values for XGraph and XDiGraphs. The weights are assigned to be 1 for Graphs and DiGraphs.

See also bidirectional_dijkstra for more information about the algorithm.

bidirectional_dijkstra(*G*, *source*, *target*)

Dijkstra's algorithm for shortest paths using bidirectional search.

Returns a two-tuple (d,p) where d is the distance and p is the path from the source to the target.

Distances are calculated as sums of weighted edges traversed.

Edges must hold numerical values for XGraph and XDiGraphs. The weights are set to 1 for Graphs and DiGraphs.

In practice bidirectional Dijkstra is much more than twice as fast as ordinary Dijkstra.

Ordinary Dijkstra expands nodes in a sphere-like manner from the source. The radius of this sphere will eventually be the length of the shortest path. Bidirectional Dijkstra will expand nodes from both the source and the target, making two spheres of half this radius. Volume of the first sphere is pi*r*r while the others are 2*pi*r/2*r/2, making up half the volume.

This algorithm is not guaranteed to work if edge weights are negative or are floating point numbers (overflows and roundoff errors can cause problems).

single_source_dijkstra_path(*G*, *source*)

Returns the shortest paths from source to all other reachable nodes in a weighted graph G. Uses Dijkstra's algorithm.

Returns a dictionary of shortest path lengths keyed by source.

Edge data must be numerical values for XGraph and XDiGraphs. The weights are assigned to be 1 for Graphs and DiGraphs.

See also single_source_dijkstra for more information about the algorithm.

single_source_dijkstra_path_length(*G*, *source*)

Returns the shortest path lengths from source to all other reachable nodes in a weighted graph G. Uses Dijkstra's algorithm.

Returns a dictionary of shortest path lengths keyed by source.

Edge data must be numerical values for XGraph and XDiGraphs. The weights are assigned to be 1 for Graphs and DiGraphs.

See also single_source_dijkstra for more information about the algorithm.

single_source_dijkstra(*G*, *source*, *target*=None)

Dijkstra's algorithm for shortest paths in a weighted graph G.

Use:

single_source_dijkstra_path() - shortest path list of nodes

single_source_dijkstra_path_length() - shortest path length

Returns a tuple of two dictionaries keyed by node. The first stores distance from the source. The second stores the path from the source to that node.

Distances are calculated as sums of weighted edges traversed. Edges must hold numerical values for XGraph and XDiGraphs. The weights are 1 for Graphs and DiGraphs.

Optional target argument stops the search when target is found.

Based on the Python cookbook recipe (119466) at
http://aspn.activestate.com/ASPN/Cookbook/Python/Recipe/119466

This algorithm is not guaranteed to work if edge weights are negative or are floating point numbers (overflows and roundoff errors can cause problems).

See also 'bidirectional_dijkstra_path'

dijkstra_predecessor_and_distance(*G*, *source*)

Same algorithm as for single_source_dijsktra, but returns two dicts representing a list of predecessors of a node and the distance to each node respectively. The list of predecessors contains more than one element only when there are more than one shortest paths to the key node.

This routine is intended for use with the betweenness centrality algorithms in centrality.py.

floyd_warshall_array(*graph*)

The Floyd-Warshall algorithm for all pairs shortest paths.

Returns a tuple (distance,path) containing two arrays of shortest distance and paths as a predecessor matrix.

This differs from floyd_warshall only in the types of the return values. Thus, path[i,j] gives the predecessor at j on a path from i to j. A value of None indicates that no path exists. A predecessor of i indicates the beginning of the path. The advantage of this implementation is that, while running time is O(n^3), running space is O(n^2).

This algorithm handles negative weights.

floyd_warshall(*G*, *huge*=`inf`)

The Floyd-Warshall algorithm for all pairs shortest paths.

Returns a tuple (distance,path) containing two dictionaries of shortest distance and predecessor paths.

This algorithm is most appropriate for dense graphs. The running time is O(n^3), and running space is O(n^2) where n is the number of nodes in G.

For sparse graphs, see

all_pairs_shortest_path all_pairs_shortest_path_length

which are based on Dijkstra's algorithm.

predecessor(*G*, *source*, *target*=None, *cutoff*=None, *return_seen*=None)

Returns dictionary of predecessors for the path from source to all nodes in G.

Optional target returns only predecessors between source and target. Cutoff is a limit on the number of hops traversed.

Example for the path graph 0-1-2-3

```
>>> G=path_graph(4)
>>> print G.nodes()
[0, 1, 2, 3]
>>> predecessor(G,0)
{0: [], 1: [0], 2: [1], 3: [2]}
```

34.2 Variables

Name	Description
__revision__	**Value:** ''
__package__	**Value:** 'networkx'

35 Package networkx.readwrite

A package for reading and writing graphs in various formats.

35.1 Modules

- **adjlist**: Read and write NetworkX graphs.
  (Section 36, p. 137)
- **edgelist**: Read and write NetworkX graphs.
  (Section 37, p. 143)
- **gml**: Read graphs in GML format.
  (Section 38, p. 146)
- **gpickle**: Read and write NetworkX graphs.
  (Section 39, p. 148)
- **graphml**: Read graphs in GraphML format.
  (Section 40, p. 150)
- **leda**: Read graphs in LEDA format.
  (Section 41, p. 151)
- **nx_yaml**: Read and write NetworkX graphs in YAML format.
  (Section 42, p. 152)
- **sparsegraph6**: Read graphs in graph6 and sparse6 format.
  (Section 43, p. 153)

35.2 Variables

Name	Description
__package__	**Value:** `'networkx.readwrite'`

36 Module networkx.readwrite.adjlist

Read and write NetworkX graphs.

Note that NetworkX graphs can contain any hashable Python object as node (not just integers and strings). So writing a NetworkX graph as a text file may not always be what you want: see write_gpickle and gread_gpickle for that case.

This module provides the following :

Adjacency list with single line per node: Useful for connected or unconnected graphs without edge data.

write_adjlist(G, path) G=read_adjlist(path)

Adjacency list with multiple lines per node: Useful for connected or unconnected graphs with or without edge data.

write_multiline_adjlist(G, path) read_multiline_adjlist(path)

Date:

Author: Aric Hagberg (hagberg@lanl.gov) Dan Schult (dschult@colgate.edu)

36.1 Functions

write_multiline_adjlist(*G*, *path*, *delimiter*=' ', *comments*='#')

Write the graph G in multiline adjacency list format to the file or file handle path.

See read_multiline_adjlist for file format details.

```
>>> write_multiline_adjlist(G,"file.adjlist")
```

path can be a filehandle or a string with the name of the file.

```
>>> fh=open("file.adjlist")
>>> write_multiline_adjlist(G,fh)
```

Filenames ending in .gz or .bz2 will be compressed.

```
>>> write_multiline_adjlist(G,"file.adjlist.gz")
```

The file will use the default text encoding on your system. It is possible to write files in other encodings by opening the file with the codecs module. See doc/examples/unicode.py for hints.

```
>>> import codecs
>>> fh=codecs.open("file.adjlist",encoding='utf=8') # use utf-8 encoding
>>> write_multiline_adjlist(G,fh)
```

read_multiline_adjlist(*path*, *comments*='#', *delimiter*=' ', *create_using*=None, *nodetype*=None, *edgetype*=None)

Read graph in multi-line adjacency list format from path.

```
>>> G=read_multiline_adjlist("file.adjlist")
```

path can be a filehandle or a string with the name of the file.

```
>>> fh=open("file.adjlist")
>>> G=read_multiline_adjlist(fh)
```

Filenames ending in .gz or .bz2 will be compressed.

```
>>> G=read_multiline_adjlist("file.adjlist.gz")
```

nodetype is an optional function to convert node strings to nodetype

For example

```
>>> G=read_multiline_adjlist("file.adjlist", nodetype=int)
```

will attempt to convert all nodes to integer type

Since nodes must be hashable, the function nodetype must return hashable types (e.g. int, float, str, frozenset - or tuples of those, etc.)

edgetype is a function to convert edge data strings to edgetype

```
>>> G=read_multiline_adjlist("file.adjlist", edgetype=int)
```

create_using is an optional networkx graph type, the default is Graph(), a simple undirected graph

```
>>> G=read_multiline_adjlist("file.adjlist", create_using=DiGraph())
```

The comments character (default='#') at the beginning of a line indicates a comment line.

The entries are separated by delimiter (default=' '). If whitespace is significant in node or edge labels you should use some other delimiter such as a tab or other symbol.

Example multiline adjlist file format:

```
# source target for Graph or DiGraph
a 2
b
c
d 1
e
```


or

```
# source target for XGraph or XDiGraph with edge data a 2 b
edge-ab-data c edge-ac-data d 1 e edge-de-data
```

write_adjlist(*G*, *path*, *comments*='#', *delimiter*=' ')

Write graph G in single-line adjacency-list format to path.

See read_adjlist for file format details.

```
>>> write_adjlist(G, "file.adjlist")
```

path can be a filehandle or a string with the name of the file.

```
>>> fh=open("file.adjlist")
>>> write_adjlist(G, fh)
```

Filenames ending in .gz or .bz2 will be compressed.

```
>>> write_adjlist(G, "file.adjlist.gz")
```

The file will use the default text encoding on your system. It is possible to write files in other encodings by opening the file with the codecs module. See doc/examples/unicode.py for hints.

```
>>> import codecs
>>> fh=codecs.open("file.adjlist",encoding='utf=8') # use utf-8 encoding
>>> write_adjlist(G,fh)
```

Does not handle data in XGraph or XDiGraph, use 'write_edgelist' or 'write_multiline_adjlist'

read_adjlist(*path*, *comments*='#', *delimiter*=' ', *create_using*=None, *nodetype*=None)

Read graph in single line adjacency list format from path.

```
>>> G=read_adjlist("file.adjlist")
```

path can be a filehandle or a string with the name of the file.

```
>>> fh=open("file.adjlist")
>>> G=read_adjlist(fh)
```

Filenames ending in .gz or .bz2 will be compressed.

```
>>> G=read_adjlist("file.adjlist.gz")
```

nodetype is an optional function to convert node strings to nodetype

For example

```
>>> G=read_adjlist("file.adjlist", nodetype=int)
```

will attempt to convert all nodes to integer type

Since nodes must be hashable, the function nodetype must return hashable types (e.g. int, float, str, frozenset - or tuples of those, etc.)

create_using is an optional networkx graph type, the default is Graph(), a simple undirected graph

```
>>> G=read_adjlist("file.adjlist", create_using=DiGraph())
```

Does not handle edge data: use 'read_edgelist' or 'read_multiline_adjlist'

The comments character (default='#') at the beginning of a line indicates a comment line.

The entries are separated by delimiter (default=' '). If whitespace is significant in node or edge labels you should use some other delimiter such as a tab or other symbol.

```
    # source target a b c d e
```

36.2 Variables

Name	Description
__credits__	**Value:** ''

continued on next page

Name	Description
__revision__	**Value:** `''`
__package__	**Value:** `'networkx.readwrite'`

37 Module networkx.readwrite.edgelist

Read and write NetworkX graphs.

Note that NetworkX graphs can contain any hashable Python object as node (not just integers and strings). So writing a NetworkX graph as a text file may not always be what you want: see write_gpickle and gread_gpickle for that case.

This module provides the following :

Edgelist format: Useful for connected graphs with or without edge data.

write_edgelist(G, path) G=read_edgelist(path)

Date:

Author: Aric Hagberg (hagberg@lanl.gov) Dan Schult (dschult@colgate.edu)

37.1 Functions

write_edgelist(*G*, *path*, *comments*='#', *delimiter*=' ')

Write graph G in edgelist format on file path.

See read_edgelist for file format details.

```
>>> write_edgelist(G, "file.edgelist")
```

path can be a filehandle or a string with the name of the file.

```
>>> fh=open("file.edgelist")
>>> write_edgelist(G,fh)
```

Filenames ending in .gz or .bz2 will be compressed.

```
>>> write_edgelist(G, "file.edgelist.gz")
```

The file will use the default text encoding on your system. It is possible to write files in other encodings by opening the file with the codecs module. See doc/examples/unicode.py for hints.

```
>>> import codecs
>>> fh=codecs.open("file.edgelist",encoding='utf=8') # use utf-8 encoding
>>> write_edgelist(G,fh)
```

read_edgelist(*path*, *comments*='#', *delimiter*=' ', *create_using*=None, *nodetype*=None, *edgetype*=None)

Read graph in edgelist format from path.

```
>>> G=read_edgelist("file.edgelist")
```

path can be a filehandle or a string with the name of the file.

```
>>> fh=open("file.edgelist")
>>> G=read_edgelist(fh)
```

Filenames ending in .gz or .bz2 will be compressed.

```
>>> G=read_edgelist("file.edgelist.gz")
```

nodetype is an optional function to convert node strings to nodetype

For example

```
>>> G=read_edgelist("file.edgelist", nodetype=int)
```

will attempt to convert all nodes to integer type

Since nodes must be hashable, the function nodetype must return hashable types (e.g. int, float, str, frozenset - or tuples of those, etc.)

create_using is an optional networkx graph type, the default is Graph(), a simple undirected graph

```
>>> G=read_edgelist("file.edgelist",create_using=DiGraph())
```

The comments character (default='#') at the beginning of a line indicates a comment line.

The entries are separated by delimiter (default=' '). If whitespace is significant in node or edge labels you should use some other delimiter such as a tab or other symbol.

Example edgelist file format:

```
# source target
a b
a c
d e
```

or for an XGraph() with edge data

source target data a b 1 a c 3.14159 d e apple

37.2 Variables

Name	Description
__credits__	**Value:** `''`
__revision__	**Value:** `''`
__package__	**Value:** `'networkx.readwrite'`

38 Module networkx.readwrite.gml

Read graphs in GML format. See http://www.infosun.fim.uni-passau.de/Graphlet/GML/gml-tr.html for format specification.

Example graphs in GML format: http://www-personal.umich.edu/~mejn/netdata/ **Author:** Aric Hagberg (hagberg@lanl.gov)

38.1 Functions

read_gml(*path*)

Read graph in GML format from path. Returns an XGraph or XDiGraph.

This doesn't implement the complete GML specification for nested attributes for graphs, edges, and nodes.

parse_gml(*lines*)

Parse GML format from string or iterable. Returns an XGraph or XDiGraph.

This doesn't implement the complete GML specification for nested attributes for graphs, edges, and nodes.

pyparse_gml()

pyparser tokenizer for GML graph format

This doesn't implement the complete GML specification for nested attributes for graphs, edges, and nodes.

write_gml(*G*, *path*)

Write the graph G in GML format to the file or file handle path.

```
>>> write_gml(G,"file.gml")
```

path can be a filehandle or a string with the name of the file.

```
>>> fh=open("file.gml")
>>> write_multiline_adjlist(G,fh)
```

Filenames ending in .gz or .bz2 will be compressed.

```
>>> write_multiline_adjlist(G,"file.gml.gz")
```

The output file will use the default text encoding on your system. It is possible to write files in other encodings by opening the file with the codecs module. See doc/examples/unicode.py for hints.

```
>>> import codecs
>>> fh=codecs.open("file.edgelist",encoding='iso8859-1')# use iso8859-1
>>> write_edgelist(G,fh)
```

GML specifications indicate that the file should only use 7bit ASCII text encoding.iso8859-1 (latin-1).

Only a single level of attributes for graphs, nodes, and edges, is supported.

38.2 Variables

Name	Description
graph	**Value:** `None`
__package__	**Value:** `'networkx.readwrite'`

39 Module networkx.readwrite.gpickle

Read and write NetworkX graphs.

Note that NetworkX graphs can contain any hashable Python object as node (not just integers and strings). So writing a NetworkX graph as a text file may not always be what you want: see write_gpickle and gread_gpickle for that case.

This module provides the following :

Python pickled format: Useful for graphs with non text representable data.

write_gpickle(G, path) read_gpickle(path)

Date:

Author: Aric Hagberg (hagberg@lanl.gov) Dan Schult (dschult@colgate.edu)

39.1 Functions

write_gpickle(*G*, *path*)

Write graph object in Python pickle format.

This will preserve Python objects used as nodes or edges.

```
>>> write_gpickle(G,"file.gpickle")
```

See cPickle.

read_gpickle(*path*)

Read graph object in Python pickle format

```
>>> G=read_gpickle("file.gpickle")
```

See cPickle.

39.2 Variables

Name	Description
__credits__	**Value:** ''
__revision__	**Value:** '$$'

continued on next page

Name	Description
__package__	**Value:** `'networkx.readwrite'`

40 Module networkx.readwrite.graphml

Read graphs in GraphML format. http://graphml.graphdrawing.org/ **Date:**

Author: Aric Hagberg (hagberg@lanl.gov)

40.1 Functions

read_graphml(*path*)

Read graph in GraphML format from path. Returns an XGraph or XDiGraph.

parse_graphml(*lines*)

Read graph in GraphML format from string. Returns an XGraph or XDiGraph.

40.2 Variables

Name	Description
__credits__	**Value:** `''`
__revision__	**Value:** `''`
__package__	**Value:** `'networkx.readwrite'`

41 Module networkx.readwrite.leda

Read graphs in LEDA format. See http://www.algorithmic-solutions.info/leda_guide/graphs/leda_native_g
Date:

Author: Aric Hagberg (hagberg@lanl.gov)

41.1 Functions

read_leda(*path*)

Read graph in GraphML format from path. Returns an XGraph or XDiGraph.

parse_leda(*lines*)

Parse LEDA.GRAPH format from string or iterable. Returns an XGraph or XDiGraph.

41.2 Variables

Name	Description
__credits__	**Value:** ''
__revision__	**Value:** ''
__package__	**Value:** 'networkx.readwrite'

42 Module networkx.readwrite.nx_yaml

Read and write NetworkX graphs in YAML format. See http://www.yaml.org for documentation. **Date:**

Author: Aric Hagberg (hagberg@lanl.gov)

42.1 Functions

write_yaml(*G*, *path*, *default_flow_style*=`False`, ***kwds*)

Write graph G in YAML text format to path.

See http://www.yaml.org

read_yaml(*path*)

Read graph from YAML format from path.

See http://www.yaml.org

42.2 Variables

Name	Description
__credits__	**Value:** `''`
__revision__	**Value:** `'$$'`
__package__	**Value:** `'networkx.readwrite'`

43 Module networkx.readwrite.sparsegraph6

Read graphs in graph6 and sparse6 format. See http://cs.anu.edu.au/~bdm/data/formats.txt
Date:

Author: Aric Hagberg (hagberg@lanl.gov)

43.1 Functions

read_graph6_list(*path*)

Read simple undirected graphs in graph6 format from path. Returns a list of Graphs, one for each line in file.

read_graph6(*path*)

Read simple undirected graphs in graph6 format from path. Returns a single Graph.

read_sparse6_list(*path*)

Read simple undirected graphs in sparse6 format from path. Returns a list of Graphs, one for each line in file.

read_sparse6(*path*)

Read simple undirected graphs in sparse6 format from path. Returns a single Graph.

graph6data(*str*)

Convert graph6 character sequence to 6-bit integers.

graph6n(*data*)

Read initial one or four-unit value from graph6 sequence. Return value, rest of seq.

parse_graph6(*str*)

Read undirected graph in graph6 format.

parse_sparse6(*str*)

Read undirected graph in sparse6 format.

43.2 Variables

Name	Description
__credits__	**Value:** `''`
__revision__	**Value:** `''`
__package__	**Value:** `'networkx.readwrite'`

44 Module networkx.release

Release data for NetworkX. **Author:** Aric Hagberg (hagberg@lanl.gov) Pieter Swart (swart@lanl.gov) Dan Schult (dschult@colgate.edu)

44.1 Variables

Name	Description
name	**Value:** `'networkx'`
version	**Value:** `'0.36'`
description	**Value:** `'Python package for creating and manipulating graphs and ...`
long_description	**Value:** `'\nNetworkX is a Python package for the creation, manipul...`
license	**Value:** `'LGPL'`
authors	**Value:** `{'Hagberg': ('Aric Hagberg', 'hagberg@lanl.gov'), 'Schult...`
url	**Value:** `'http://networkx.lanl.gov/'`
download_url	**Value:** `'http://networkx.lanl.gov/download'`
platforms	**Value:** `['Linux', 'Mac OSX', 'Windows XP/2000/NT']`
keywords	**Value:** `['Networks', 'Graph Theory', 'Mathematics', 'network', 'g...`
classifiers	**Value:** `['Development Status :: 4 - Beta', 'Intended Audience :: ...`
date	**Value:** `'Tue Jun 16 14:09:53 2009'`
__package__	**Value:** `'networkx'`

45 Module networkx.search

Search algorithms.

See also networkx.path. **Date:**

Author: Eben Kenah (ekenah@t7.lanl.gov) Aric Hagberg (hagberg@lanl.gov)

45.1 Functions

dfs_preorder(*G*, *source*=`None`, *reverse_graph*=`False`)

Return list of nodes connected to source in DFS preorder. Traverse the graph G with depth-first-search from source. Non-recursive algorithm.

dfs_postorder(*G*, *source*=`None`, *reverse_graph*=`False`)

Return list of nodes connected to source in DFS preorder. Traverse the graph G with depth-first-search from source. Non-recursive algorithm.

dfs_tree(*G*, *source*=`None`, *reverse_graph*=`False`)

Return directed graph (tree) of depth-first-search with root at source. If the graph is disconnected, return a disconnected graph (forest).

dfs_predecessor(*G*, *source*=`None`, *reverse_graph*=`False`)

Return predecessors of depth-first-search with root at source.

dfs_successor(*G*, *source*=`None`, *reverse_graph*=`False`)

Return succesors of depth-first-search with root at source.

45.2 Variables

Name	Description
__credits__	**Value:** `''`
__revision__	**Value:** `''`
__package__	**Value:** `'networkx'`

46 Module networkx.spectrum

Laplacian, adjacency matrix, and spectrum of graphs.

Needs numpy array package: numpy.scipy.org. **Author:** Aric Hagberg (hagberg@lanl.gov) Pieter Swart (swart@lanl.gov) Dan Schult(dschult@colgate.edu)

46.1 Functions

adj_matrix(*G*, *nodelist*=`None`)

Return adjacency matrix of graph as a numpy matrix.

This just calls networkx.convert.to_numpy_matrix.

If you want a pure python adjacency matrix represntation try networkx.convert.to_dict_of_dicts with weighted=False, which will return a dictionary-of-dictionaries format that can be addressed as a sparse matrix.

laplacian(*G*, *nodelist*=`None`)

Return standard combinatorial Laplacian of G as a numpy matrix.

Return the matrix L = D - A, where

> D is the diagonal matrix in which the i'th entry is the degree of node i A is the adjacency matrix.

normalized_laplacian(*G*, *nodelist*=`None`)

Return normalized Laplacian of G as a numpy matrix.

See Spectral Graph Theory by Fan Chung-Graham. CBMS Regional Conference Series in Mathematics, Number 92, 1997.

laplacian_spectrum(*G*)

Return eigenvalues of the Laplacian of G

adjacency_spectrum(*G*)

Return eigenvalues of the adjacency matrix of G

combinatorial_laplacian(*G*, *nodelist*=`None`)

Return standard combinatorial Laplacian of G as a numpy matrix.

Return the matrix L = D - A, where

D is the diagonal matrix in which the i'th entry is the degree of node i A is the adjacency matrix.

generalized_laplacian(*G*, *nodelist*=`None`)

Return normalized Laplacian of G as a numpy matrix.

See Spectral Graph Theory by Fan Chung-Graham. CBMS Regional Conference Series in Mathematics, Number 92, 1997.

46.2 Variables

Name	Description
__package__	**Value:** `'networkx'`

47 Package networkx.tests

47.1 Modules

- **benchmark** *(Section 48, p. 161)*
- **drawing** *(Section 49, p. 163)*
- **generators** *(Section 50, p. 164)*
- **readwrite** *(Section 51, p. 165)*
- **test** *(Section 52, p. 166)*

47.2 Variables

Name	Description
__package__	**Value:** `'networkx.tests'`

48 Module networkx.tests.benchmark

48.1 Variables

Name	Description
__package__	**Value:** `'networkx.tests'`

48.2 Class Benchmark

object
└
networkx.tests.benchmark.Benchmark

Benchmark a method or simple bit of code using different Graph classes. If the test code is the same for each graph class, then you can set it during instantiation through the argument test_string. The argument test_string can also be a tuple of test code and setup code. The code is entered as a string valid for use with the timeit module.

Example: >>> b=Benchmark(['Graph','XGraph']) >>> b['Graph']=('G.add_nodes_from(nlist)','nlist=ra
>>> b.run()

48.2.1 Methods

__init__(*self*, *graph_classes*, *title*=`''`, *test_string*=`None`, *runs*=`3`, *reps*=`1000`)

x.__init__(...) initializes x; see x.__class__.__doc__ for signature Overrides: object.__init__ extit(inherited documentation)

__setitem__(*self*, *graph_class*, *(test_str, setup_str)*)

Set a simple bit of code and setup string for the test. Use this for cases where the code differs from one class to another.

run(*self*)

Run the benchmark for each class and print results.

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __new__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

48.2.2 Properties

Name	Description
Inherited from object	
__class__	

49 Package networkx.tests.drawing

49.1 Variables

Name	Description
__package__	**Value:** `None`

50 Package networkx.tests.generators

50.1 Variables

Name	Description
__package__	**Value:** `None`

51 Package networkx.tests.readwrite

51.1 Variables

Name	Description
__package__	**Value:** `None`

52 Module networkx.tests.test

52.1 Functions

all()

run()

52.2 Variables

Name	Description
__package__	**Value:** `'networkx.tests'`

53 Module networkx.threshold

Threshold Graphs - Creation, manipulation and identification. **Version:** $Revision: 1049 $

Date: $Date: 2005-06-17 08:06:22 -0600 (Fri, 17 Jun 2005) $

Author: Aric Hagberg (hagberg@lanl.gov) Pieter Swart (swart@lanl.gov) Dan Schult (dschult@colgate.edu)

53.1 Functions

is_threshold_graph(*G*)

Returns True if G is a threshold graph.

is_threshold_sequence(*degree_sequence*)

Returns True if the sequence is a threshold degree seqeunce.

Uses the property that a threshold graph must be constructed by adding either dominating or isolated nodes. Thus, it can be deconstructed iteratively by removing a node of degree zero or a node that connects to the remaining nodes. If this deconstruction failes then the sequence is not a threshold sequence.

creation_sequence(*degree_sequence*, *with_labels*=`False`, *compact*=`False`)

Determines the creation sequence for the given threshold degree sequence.

The creation sequence is a list of single characters 'd' or 'i': 'd' for dominating or 'i' for isolated vertices. Dominating vertices are connected to all vertices present when it is added. The first node added is by convention 'd'. This list can be converted to a string if desired using “”.join(cs)

If with_labels==True: Returns a list of 2-tuples containing the vertex number and a character 'd' or 'i' which describes the type of vertex.

If compact==True: Returns the creation sequence in a compact form that is the number of 'i's and 'd's alternating. Examples: [1,2,2,3] represents d,i,i,d,d,i,i,i [3,1,2] represents d,d,d,i,d,d

Notice that the first number is the first vertex to be used for construction and so is always 'd'.

with_labels and compact cannot both be True.

Returns None if the sequence is not a threshold sequence

make_compact(*creation_sequence*)

Returns the creation sequence in a compact form that is the number of 'i's and 'd's alternating. Examples: [1,2,2,3] represents d,i,i,d,d,i,i,i. [3,1,2] represents d,d,d,i,d,d. Notice that the first number is the first vertex to be used for construction and so is always 'd'.

Labeled creation sequences lose their labels in the compact representation.

uncompact(*creation_sequence*)

Converts a compact creation sequence for a threshold graph to a standard creation sequence (unlabeled). If the creation_sequence is already standard, return it. See creation_sequence.

creation_sequence_to_weights(*creation_sequence*)

Returns a list of node weights which create the threshold graph designated by the creation sequence. The weights are scaled so that the threshold is 1.0. The order of the nodes is the same as that in the creation sequence.

weights_to_creation_sequence(*weights*, *threshold*=`1`, *with_labels*=`False`, *compact*=`False`)

Returns a creation sequence for a threshold graph determined by the weights and threshold given as input. If the sum of two node weights is greater than the threshold value, an edge is created between these nodes.

The creation sequence is a list of single characters 'd' or 'i': 'd' for dominating or 'i' for isolated vertices. Dominating vertices are connected to all vertices present when it is added. The first node added is by convention 'd'.

If with_labels==True: Returns a list of 2-tuples containing the vertex number and a character 'd' or 'i' which describes the type of vertex.

If compact==True: Returns the creation sequence in a compact form that is the number of 'i's and 'd's alternating. Examples: [1,2,2,3] represents d,i,i,d,d,i,i,i [3,1,2] represents d,d,d,i,d,d

Notice that the first number is the first vertex to be used for construction and so is always 'd'.

with_labels and compact cannot both be True.

threshold_graph(*creation_sequence*)

Create a threshold graph from the creation sequence or compact creation_sequence.

The input sequence can be a

creation sequence (e.g. ['d','i','d','d','d','i']) labeled creation sequence (e.g. [(0,'d'),(2,'d'),(1,'i')]) compact creation sequence (e.g. [2,1,1,2,0])

Use cs=creation_sequence(degree_sequence,labeled=True) to convert a degree sequence to a creation sequence.

Returns None if the sequence is not valid

find_alternating_4_cycle(*G*)

Returns False if there aren't any alternating 4 cycles. Otherwise returns the cycle as [a,b,c,d] where (a,b) and (c,d) are edges and (a,c) and (b,d) are not.

find_threshold_graph(*G*)

Return a threshold subgraph that is close to largest in G. The threshold graph will contain the largest degree node in G.

find_creation_sequence(*G*)

Find a threshold subgraph that is close to largest in G. Returns the labeled creation sequence of that threshold graph.

triangles(*creation_sequence*)

Compute number of triangles in the threshold graph with the given creation sequence.

triangle_sequence(*creation_sequence*)

Return triangle sequence for the given threshold graph creation sequence.

cluster_sequence(*creation_sequence*)

Return cluster sequence for the given threshold graph creation sequence.

degree_sequence(*creation_sequence*)

Return degree sequence for the threshold graph with the given creation sequence

density(*creation_sequence*)

Return the density of the graph with this creation_sequence. The density is the fraction of possible edges present.

degree_correlation(*creation_sequence*)

Return the degree-degree correlation over all edges.

shortest_path(*creation_sequence*, *u*, *v*)

Find the shortest path between u and v in a threshold graph G with the given creation_sequence.

For an unlabeled creation_sequence, the vertices u and v must be integers in (0,len(sequence)) refering to the position of the desired vertices in the sequence.

For a labeled creation_sequence, u and v are labels of veritices.

Use cs=creation_sequence(degree_sequence,with_labels=True) to convert a degree sequence to a creation sequence.

Returns a list of vertices from u to v. Example: if they are neighbors, it returns [u,v]

shortest_path_length(*creation_sequence*, *i*)

Return the shortest path length from indicated node to every other node for the threshold graph with the given creation sequence. Node is indicated by index i in creation_sequence unless creation_sequence is labeled in which case, i is taken to be the label of the node.

Paths lengths in threshold graphs are at most 2. Length to unreachable nodes is set to -1.

betweenness_sequence(*creation_sequence*, *normalized*=`True`)

Return betweenness for the threshold graph with the given creation sequence. The result is unscaled. To scale the values to the iterval [0,1] divide by (n-1)*(n-2).

eigenvectors(*creation_sequence*)

Return a 2-tuple of Laplacian eigenvalues and eigenvectors for the threshold network with creation_sequence. The first value is a list of eigenvalues. The second value is a list of eigenvectors. The lists are in the same order so corresponding eigenvectors and eigenvalues are in the same position in the two lists.

Notice that the order of the eigenvalues returned by eigenvalues(cs) may not correspond to the order of these eigenvectors.

spectral_projection(*u*, *eigenpairs*)

Returns the coefficients of each eigenvector in a projection of the vector u onto the normalized eigenvectors which are contained in eigenpairs.

eigenpairs should be a list of two objects. The first is a list of eigenvalues and the second a list of eigenvectors. The eigenvectors should be lists.

There's not a lot of error checking on lengths of arrays, etc. so be careful.

eigenvalues(*creation_sequence*)

Return sequence of eigenvalues of the Laplacian of the threshold graph for the given creation_sequence.

Based on the Ferrer's diagram method. The spectrum is integral and is the conjugate of the degree sequence.

See:

```
@Article{degree-merris-1994,
 author =          {Russel Merris},
 title =   {Degree maximal graphs are Laplacian inte-
gral},
 journal =         {Linear Algebra Appl.},
 year =    {1994},
 volume =          {199},
 pages =   {381--389},
}
```

random_threshold_sequence(*n*, *p*, *seed*=None)

Create a random threshold sequence of size n. A creation sequence is built by randomly choosing d's with probabiliy p and i's with probability 1-p.

```
>>> s=random_threshold_sequence(10,0.5)
```

returns a threshold sequence of length 10 with equal probably of an i or a d at each position.

A "random" threshold graph can be built with

```
>>> G=threshold_graph(random_threshold_sequence(10,0.5))
```

right_d_threshold_sequence(*n*, *m*)

Create a skewed threshold graph with a given number of vertices (n) and a given number of edges (m).

The routine returns an unlabeled creation sequence for the threshold graph.

FIXME: describe algorithm

left_d_threshold_sequence(*n*, *m*)

Create a skewed threshold graph with a given number of vertices (n) and a given number of edges (m).

The routine returns an unlabeled creation sequence for the threshold graph.

FIXME: describe algorithm

swap_d(*cs*, *p_split*=`1.0`, *p_combine*=`1.0`, *seed*=`None`)

Perform a "swap" operation on a threshold sequence.

The swap preserves the number of nodes and edges in the graph for the given sequence. The resulting sequence is still a threshold sequence.

Perform one split and one combine operation on the 'd's of a creation sequence for a threshold graph. This operation maintains the number of nodes and edges in the graph, but shifts the edges from node to node maintaining the threshold quality of the graph.

53.2 Variables

Name	Description
__credits__	**Value:** `''`
__package__	**Value:** `'networkx'`

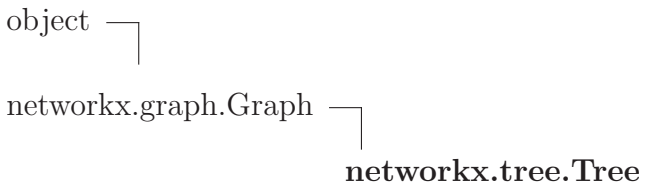
54 Module networkx.tree

EXPERIMENTAL: Base classes for trees and forests. **Author:** Aric Hagberg (hagberg@lanl.gov)

54.1 Variables

Name	Description
__package__	**Value:** `'networkx'`

54.2 Class Tree



Known Subclasses: networkx.tree.DirectedForest, networkx.tree.DirectedTree, networkx.tree.Forest, networkx.tree.RootedTree

A free (unrooted) tree.

54.2.1 Methods

__init__(*self*, *data*=`None`, ***kwds*)

Initialize Graph.

```
>>> G=Graph(name="empty")
```

creates empty graph G with G.name="empty" Overrides: object.__init__ extit(inherited documentation)

add_node(*self*, *n*)

Add a single node n to the graph.

The node n can be any hashable object except None.

A hashable object is one that can be used as a key in a Python dictionary. This includes strings, numbers, tuples of strings and numbers, etc. On many platforms this also includes mutables such as Graphs e.g., though one should be careful the hash doesn't change on mutables.

Example:

```
>>> from networkx import *
>>> G=Graph()
>>> K3=complete_graph(3)
>>> G.add_node(1)
>>> G.add_node('Hello')
>>> G.add_node(K3)
>>> G.number_of_nodes()
3
```

Overrides: networkx.graph.Graph.add_node extit(inherited documentation)

add_nodes_from(*self*, *nbunch*)

Add multiple nodes to the graph.

nlist: A container of nodes that will be iterated through once (thus it should be an iterator or be iterable). Each element of the container should be a valid node type: any hashable type except None. See add_node for details.

Examples:

```
>>> from networkx import *
>>> G=Graph()
>>> K3=complete_graph(3)
>>> G.add_nodes_from('Hello')
>>> G.add_nodes_from(K3)
>>> sorted(G.nodes())
[0, 1, 2, 'H', 'e', 'l', 'o']
```

Overrides: networkx.graph.Graph.add_nodes_from extit(inherited documentation)

delete_node(*self*, *n*)

Delete node n from graph. Attempting to delete a non-existent node will raise an exception. Overrides: networkx.graph.Graph.delete_node extit(inherited documentation)

delete_nodes_from(*self*, *nbunch*)

Remove nodes in nlist from graph.

nlist: an iterable or iterator containing valid node names.

Attempting to delete a non-existent node will raise an exception. This could mean some nodes got deleted and other valid nodes did not. Overrides: networkx.graph.Graph.delete_nodes_from extit(inherited documentation)

add_edge(*self*, *u*, *v*=`None`)

Add a single edge (u,v) to the graph.

>> G.add_edge(u,v) and >>> G.add_edge((u,v)) are equivalent forms of adding a single edge between nodes u and v. The nodes u and v will be automatically added if not already in the graph. They must be a hashable (except None) Python object.

The following examples all add the edge (1,2) to graph G.

```
>>> G=Graph()
>>> G.add_edge( 1, 2 )           # explicit two node form
>>> G.add_edge( (1,2) )          # single edge as tuple of two nodes
>>> G.add_edges_from( [(1,2)] ) # add edges from iterable container
```

Overrides: networkx.graph.Graph.add_edge extit(inherited documentation)

add_edges_from(*self*, *ebunch*)

Add all the edges in ebunch to the graph.

ebunch: Container of 2-tuples (u,v). The container must be iterable or an iterator. It is iterated over once. Adding the same edge twice has no effect and does not raise an exception. Overrides: networkx.graph.Graph.add_edges_from extit(inherited documentation)

delete_edge(*self*, *u*, *v*=None)

Delete the single edge (u,v).

Can be used in two basic forms: >>> G.delete_edge(u,v) and >> G.delete_edge((u,v)) are equivalent ways of deleting a single edge between nodes u and v.

Return without complaining if the nodes or the edge do not exist. Overrides: networkx.graph.Graph.delete_edge extit(inherited documentation)

delete_edges_from(*self*, *ebunch*)

Delete the edges in ebunch from the graph.

ebunch: an iterator or iterable of 2-tuples (u,v).

Edges that are not in the graph are ignored. Overrides: networkx.graph.Graph.delete_edges_from extit(inherited documentation)

add_leaf(*self*, *u*, *v*=None)

delete_leaf(*self*, *u*, *v*=None)

add_leaves_from(*self*, *ebunch*)

delete_leaves_from(*self*, *ebunch*)

union_sub(*self*, *T1*, ***kwds*)

Polymorphic helper method for Graph.union().

Required keywords: v_from and v_to, where v_from is the node in self to which v_to should be attached as child.

union_sub_tree_helper(*self*, *T1*, *parent*, *grandparent*=None)

Inherited from networkx.graph.Graph(Section 28.2)

__contains__(), __getitem__(), __iter__(), __len__(), __str__(), add_cycle(), add_path(), clear(), copy(), degree(), degree_iter(), edge_boundary(), edges(), edges_iter(), get_edge(), has_edge(), has_neighbor(), has_node(), info(), is_directed(), neighbors(), neighbors_iter(), node_boundary(), nodes(), nodes_iter(), number_of_edges(), number_of_nodes(), order(), prepare_nbunch(), size(), subgraph(), to_directed(),

to_undirected()

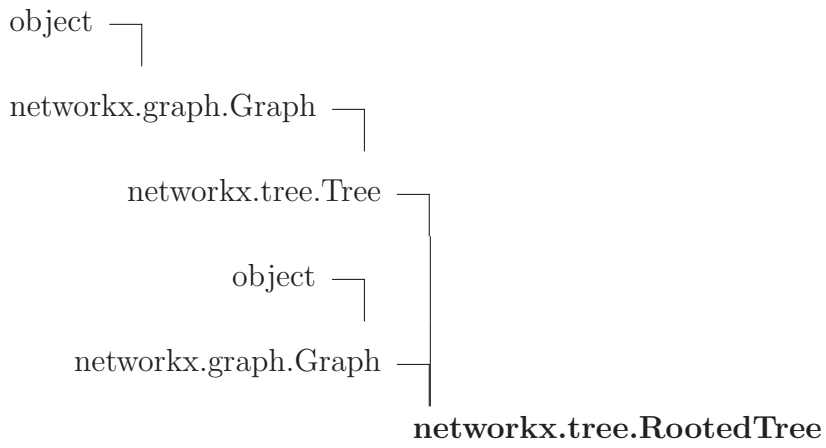
Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __new__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __subclasshook__()

54.2.2 Properties

Name	Description
Inherited from object	
__class__	

54.3 Class RootedTree



A rooted tree.

54.3.1 Methods

__init__(*self*, *root*, *data*=None, ***kwds*)

Initialize Graph.

```
>>> G=Graph(name="empty")
```

creates empty graph G with G.name="empty" Overrides: object.__init__ extit(inherited documentation)

delete_node(*self*, *n*)

Delete node n from graph. Attempting to delete a non-existent node will raise an exception. Overrides: networkx.graph.Graph.delete_node extit(inherited documentation)

add_edge(*self*, *u*, *v*=None)

Add a single edge (u,v) to the graph.

>> G.add_edge(u,v) and >>> G.add_edge((u,v)) are equivalent forms of adding a single edge between nodes u and v. The nodes u and v will be automatically added if not already in the graph. They must be a hashable (except None) Python object.

The following examples all add the edge (1,2) to graph G.

```
>>> G=Graph()
>>> G.add_edge( 1, 2 )           # explicit two node form
>>> G.add_edge( (1,2) )          # single edge as tuple of two nodes
>>> G.add_edges_from( [(1,2)] ) # add edges from iterable container
```

Overrides: networkx.graph.Graph.add_edge extit(inherited documentation)

parent(*self*, *u*)

children(*self*, *u*)

root_tree(*self*, *root*)

Inherited from networkx.tree.Tree(Section 54.2)

add_edges_from(), add_leaf(), add_leaves_from(), add_node(), add_nodes_from(), delete_edge(), delete_edges_from(), delete_leaf(), delete_leaves_from(), delete_nodes_from(), union_sub(), union_sub_tree_helper()

Inherited from networkx.graph.Graph(Section 28.2)

__contains__(), __getitem__(), __iter__(), __len__(), __str__(), add_cycle(), add_path(), clear(), copy(), degree(), degree_iter(), edge_boundary(), edges(), edges_iter(), get_edge(), has_edge(), has_neighbor(), has_node(), info(), is_directed(), neighbors(), neighbors_iter(), node_boundary(), nodes(), nodes_iter(), number_of_edges(), number_of_nodes(), order(), prepare_nbunch(), size(), subgraph(), to_directed(), to_undirected()

Inherited from object

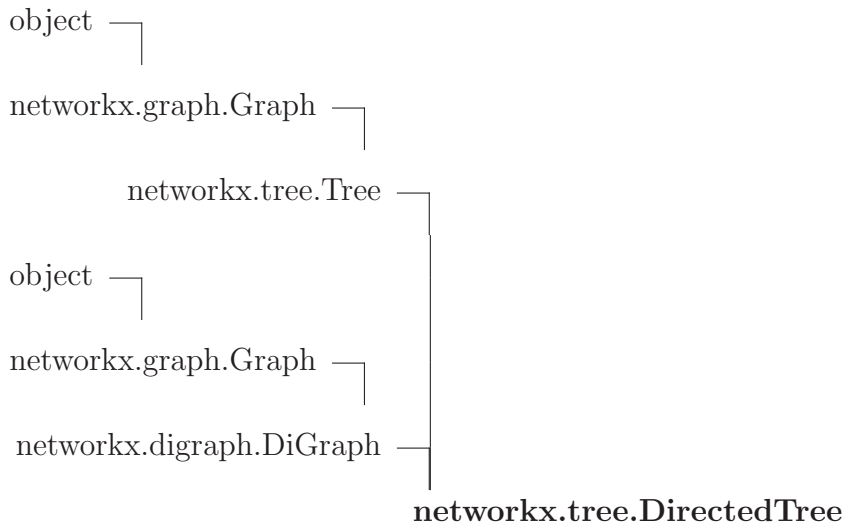
__delattr__(), __format__(), __getattribute__(), __hash__(), __new__(), __reduce__(),

__reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __subclasshook__()

54.3.2 Properties

Name	Description
Inherited from object	
__class__	

54.4 Class DirectedTree



A directed tree.

54.4.1 Methods

__init__(*self*, *data*=None, ***kwds*)

Initialize Graph.

```
>>> G=Graph(name="empty")
```

creates empty graph G with G.name="empty" Overrides: object.__init__ extit(inherited documentation)

delete_node(*self*, *n*)

Delete node n from the digraph. Attempting to delete a non-existent node will raise a NetworkXError. Overrides: networkx.graph.Graph.delete_node extit(inherited documentation)

add_edge(*self*, *u*, *v*=None)

Add a single directed edge (u,v) to the digraph.

>> G.add_edge(u,v) and >>> G.add_edge((u,v)) are equivalent forms of adding a single edge between nodes u and v. The nodes u and v will be automatically added if not already in the graph. They must be a hashable (except None) Python object.

For example, the following examples all add the edge (1,2) to the digraph G.

```
>>> G=DiGraph()
>>> G.add_edge( 1, 2 )           # explicit two node form
>>> G.add_edge( (1,2) )          # single edge as tuple of two nodes
>>> G.add_edges_from( [(1,2)] ) # list of edges form
```

Overrides: networkx.graph.Graph.add_edge extit(inherited documentation)

delete_edge(*self*, *u*, *v*=None)

Delete the single directed edge (u,v) from the digraph.

Can be used in two basic forms >>> G.delete_edge(u,v) and G.delete_edge((u,v)) are equivalent ways of deleting a directed edge u->v.

If the edge does not exist return without complaining. Overrides: networkx.graph.Graph.delete_edge extit(inherited documentation)

Inherited from networkx.tree.Tree(Section 54.2)

add_edges_from(), add_leaf(), add_leaves_from(), add_node(), add_nodes_from(), delete_edges_from(), delete_leaf(), delete_leaves_from(), delete_nodes_from(), union_sub(), union_sub_tree_helper()

Inherited from networkx.digraph.DiGraph(Section 9.2)

clear(), copy(), degree_iter(), edges_iter(), in_degree(), in_degree_iter(), in_edges(), in_edges_iter(), in_neighbors(), is_directed(), neighbors(), neighbors_iter(), out_degree(), out_degree_iter(), out_edges(), out_edges_iter(), out_neighbors(), predecessors(), predecessors_iter(), reverse(), subgraph(), successors(), successors_iter(), to_directed(), to_undirected()

Inherited from networkx.graph.Graph(Section 28.2)

__contains__(), __getitem__(), __iter__(), __len__(), __str__(), add_cycle(), add_path(), degree(), edge_boundary(), edges(), get_edge(), has_edge(), has_neighbor(), has_node(), info(), node_boundary(), nodes(), nodes_iter(), number_of_edges(), number_of_nodes(), order(), prepare_nbunch(), size()

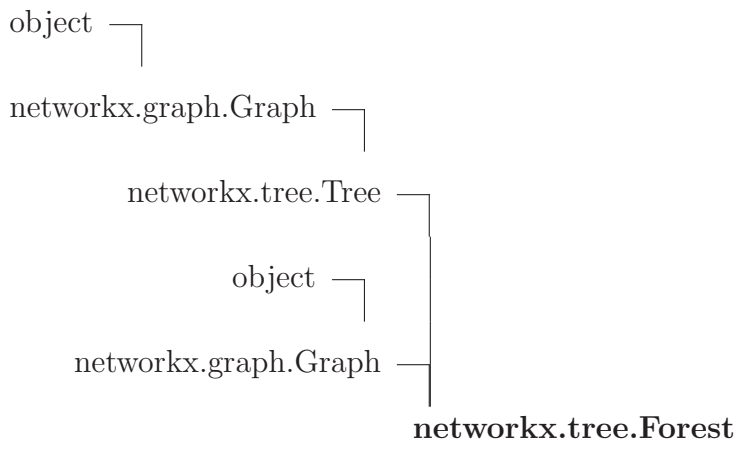
Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __new__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __subclasshook__()

54.4.2 Properties

Name	Description
Inherited from object	
__class__	

54.5 Class Forest



A forest.

54.5.1 Methods

__init__(*self*, *data*=None, ***kwds*)

Initialize Graph.

```
>>> G=Graph(name="empty")
```

creates empty graph G with G.name="empty" Overrides: object.__init__ extit(inherited documentation)

add_node(*self*, *n*)

Add a single node n to the graph.

The node n can be any hashable object except None.

A hashable object is one that can be used as a key in a Python dictionary. This includes strings, numbers, tuples of strings and numbers, etc. On many platforms this also includes mutables such as Graphs e.g., though one should be careful the hash doesn't change on mutables.

Example:

```
>>> from networkx import *
>>> G=Graph()
>>> K3=complete_graph(3)
>>> G.add_node(1)
>>> G.add_node('Hello')
>>> G.add_node(K3)
>>> G.number_of_nodes()
3
```

Overrides: networkx.graph.Graph.add_node extit(inherited documentation)

delete_node(*self*, *n*)

Delete node n from graph. Attempting to delete a non-existent node will raise an exception. Overrides: networkx.graph.Graph.delete_node extit(inherited documentation)

add_edge(*self*, *u*, *v*=None)

Add a single edge (u,v) to the graph.

>> G.add_edge(u,v) and >>> G.add_edge((u,v)) are equivalent forms of adding a single edge between nodes u and v. The nodes u and v will be automatically added if not already in the graph. They must be a hashable (except None) Python object.

The following examples all add the edge (1,2) to graph G.

```
>>> G=Graph()
>>> G.add_edge( 1, 2 )           # explicit two node form
>>> G.add_edge( (1,2) )          # single edge as tuple of two nodes
>>> G.add_edges_from( [(1,2)] ) # add edges from iterable container
```

Overrides: networkx.graph.Graph.add_edge extit(inherited documentation)

delete_edge(*self*, *u*, *v*=None)

Delete the single edge (u,v).

Can be used in two basic forms: >>> G.delete_edge(u,v) and >> G.delete_edge((u,v)) are equivalent ways of deleting a single edge between nodes u and v.

Return without complaining if the nodes or the edge do not exist. Overrides: networkx.graph.Graph.delete_edge extit(inherited documentation)

tree(*self*, *n*=None)

Return tree containing node n. If no node is specified return list of all trees in forest.

tree_nodes(*self*, *n*=None)

Return tree containing node n. If no node is specified return list of all trees in forest.

Inherited from networkx.tree.Tree(Section 54.2)

add_edges_from(), add_leaf(), add_leaves_from(), add_nodes_from(), delete_edges_from(), delete_leaf(), delete_leaves_from(), delete_nodes_from(), union_sub(), union_sub_tree_helper()

Inherited from networkx.graph.Graph(Section 28.2)

__contains__(), __getitem__(), __iter__(), __len__(), __str__(), add_cycle(), add_path(), clear(), copy(), degree(), degree_iter(), edge_boundary(), edges(), edges_iter(), get_edge(), has_edge(), has_neighbor(), has_node(), info(), is_directed(), neighbors(), neighbors_iter(), node_boundary(), nodes(), nodes_iter(), number_of_edges(), number_of_nodes(), order(), prepare_nbunch(), size(), subgraph(), to_directed(), to_undirected()

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __new__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __subclasshook__()

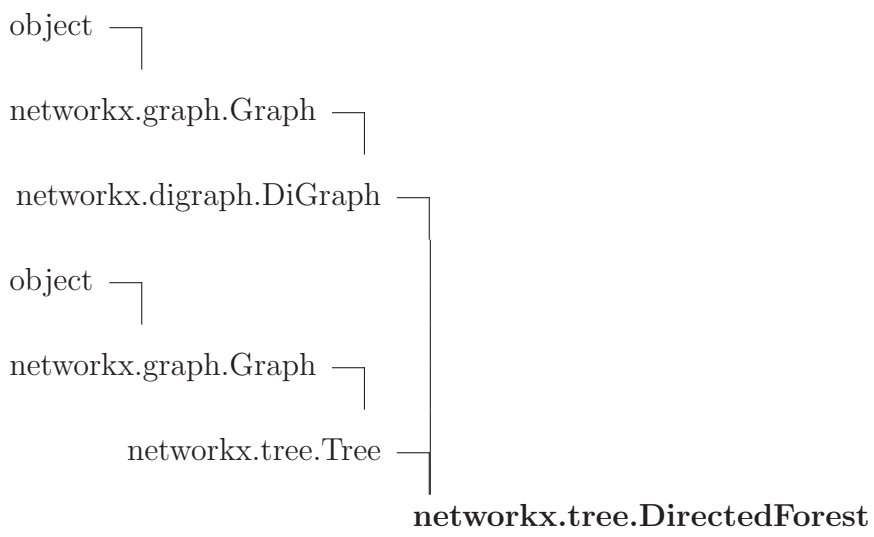
54.5.2 Properties

Name	Description
Inherited from object	

continued on next page

Name	Description
__class__	

54.6 Class DirectedForest



54.6.1 Methods

Inherited from networkx.digraph.DiGraph(Section 9.2)

__init__(), add_edge(), add_edges_from(), add_node(), add_nodes_from(), clear(), copy(), degree_iter(), delete_edge(), delete_edges_from(), delete_node(), delete_nodes_from(), edges_iter(), in_degree(), in_degree_iter(), in_edges(), in_edges_iter(), in_neighbors(), is_directed(), neighbors(), neighbors_iter(), out_degree(), out_degree_iter(), out_edges(), out_edges_iter(), out_neighbors(), predecessors(), predecessors_iter(), reverse(), subgraph(), successors(), successors_iter(), to_directed(), to_undirected()

Inherited from networkx.tree.Tree(Section 54.2)

add_leaf(), add_leaves_from(), delete_leaf(), delete_leaves_from(), union_sub(), union_sub_tree_helper()

Inherited from networkx.graph.Graph(Section 28.2)

__contains__(), __getitem__(), __iter__(), __len__(), __str__(), add_cycle(), add_path(), degree(), edge_boundary(), edges(), get_edge(), has_edge(), has_neighbor(), has_node(), info(), node_boundary(), nodes(), nodes_iter(), number_of_edges(), number_of_nodes(), order(), prepare_nbunch(), size()

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __new__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __subclasshook__()

54.6.2 Properties

Name	Description
Inherited from object	
__class__	

55 Module networkx.utils

Utilities for networkx package **Date:** $Date: 2005-06-15 08:30:40 -0600 (Wed, 15 Jun 2005) $

Author: Aric Hagberg (hagberg@lanl.gov) Dan Schult(dschult@colgate.edu)

55.1 Functions

is_singleton(*obj*)

Is string_like or not iterable.

is_string_like(*obj*)

Check if obj is string.

iterable(*obj*)

Return True if obj is iterable with a well-defined len()

flatten(*obj*, *result*=`None`)

Return flattened version of (possibly nested) iterable obj.

iterable_to_string(*obj*, *sep*=`''`)

Return string obtained by concatenating the string representation of each element of an iterable obj, with an optional internal string separator specified.

is_list_of_ints(*intlist*)

Return True if list is a list of ints.

scipy_pareto_sequence(*n*, *exponent*=`1.0`)

Return sample sequence of length n from a Pareto distribution.

scipy_powerlaw_sequence(*n*, *exponent*=`2.0`)

Return sample sequence of length n from a power law distribution.

scipy_poisson_sequence(*n*, *mu*=`1.0`)

Return sample sequence of length n from a Poisson distribution.

scipy_uniform_sequence(*n*)

Return sample sequence of length n from a uniform distribution.

scipy_discrete_sequence(*n*, *distribution*=`False`)

Return sample sequence of length n from a given discrete distribution

distribution=histogram of values, will be normalized

gsl_pareto_sequence(*n*, *exponent*=`1.0`, *scale*=`1.0`, *seed*=`None`)

Return sample sequence of length n from a Pareto distribution.

gsl_powerlaw_sequence(*n*, *exponent*=`2.0`, *scale*=`1.0`, *seed*=`None`)

Return sample sequence of length n from a power law distribution.

gsl_poisson_sequence(*n*, *mu*=`1.0`, *seed*=`None`)

Return sample sequence of length n from a Poisson distribution.

gsl_uniform_sequence(*n*, *seed*=`None`)

Return sample sequence of length n from a uniform distribution.

pareto_sequence(*n*, *exponent*=`1.0`)

Return sample sequence of length n from a Pareto distribution.

powerlaw_sequence(*n*, *exponent*=`2.0`)

Return sample sequence of length n from a power law distribution.

uniform_sequence(*n*)

Return sample sequence of length n from a uniform distribution.

cumulative_distribution(*distribution*)

Return normalized cumulative distribution from discrete distribution.

discrete_sequence(*n*, *distribution*=`None`, *cdistribution*=`None`)

Return sample sequence of length n from a given discrete distribution or discrete cumulative distribution.

One of the following must be specified.

distribution = histogram of values, will be normalized

cdistribution = normalized discrete cumulative distribution

55.2 Variables

Name	Description
__credits__	**Value:** `''`
__revision__	**Value:** `'$Revision: 1029 $'`

continued on next page

Name	Description
__package__	**Value:** `'networkx'`

56 Module networkx.xdigraph

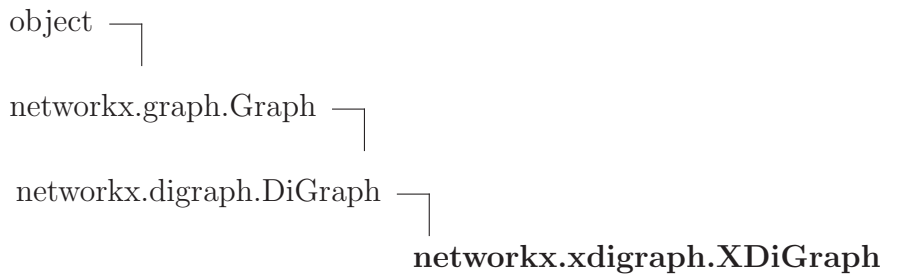
Base class for XDiGraph.

XDiGraph allows directed graphs with self-loops, multiple edges, arbitrary (hashable) objects as nodes, and arbitrary objects associated with edges. **Author:** Aric Hagberg (hagberg@lanl.gov) Pieter Swart (swart@lanl.gov) Dan Schult(dschult@colgate.edu)

56.1 Variables

Name	Description
__package__	**Value:** `'networkx'`

56.2 Class XDiGraph



Digraphs with (optional) self-loops, (optional) multiple edges, arbitrary (hashable) objects as nodes, and arbitrary objects associated with edges.

An XDiGraph edge is uniquely specified by a 3-tuple e=(n1,n2,x), where n1 and n2 are (hashable) objects (nodes) and x is an arbitrary (and not necessarily unique) object associated with that edge.

See the documentation of XGraph for the use of the optional parameters selfloops (defaults is False) and multiedges (default is False).

XDiGraph inherits from DiGraph, with all purely node-specific methods identical to those of DiGraph. XDiGraph edges are identical to XGraph edges, except that they are directed rather than undirected. XDiGraph replaces the following DiGraph methods:

- __init__: read multiedges and selfloops optional args.
- add_edge
- add_edges_from
- delete_edge

- delete_edges_from
- has_edge
- has_predecessor
- has_successor
- get_edge
- edges_iter
- in_edges_iter
- out_edges_iter
- neighbors_iter
- successors_iter
- predecessors_iter
- degree_iter
- out_degree_iter
- in_degree_iter
- subgraph
- copy
- to_undirected
- reverse

XDiGraph also adds the following methods to those of DiGraph:

- allow_selfloops
- remove_all_selfloops
- ban_selfloops
- nodes_with_selfloops
- self_loop_edges
- number_of_selfloops
- delete_multiedge
- allow_multiedges
- ban_multiedges

- remove_all_multiedges

While XDiGraph does not inherit from XGraph, we compare them here. XDigraph adds the following methods to those of XGraph:

- has_successor
- successors
- successors_iter
- has_predecessor
- predecessors
- predecessors_iter
- out_degree
- out_degree_iter
- in_degree
- in_degree_iter
- reverse

56.2.1 Methods

__init__(*self*, *data*=`None`, *name*=`''`, *selfloops*=`False`, *multiedges*=`False`)

Initialize XDiGraph.

Optional arguments:: name: digraph name (default="No Name") selfloops: if True then selfloops are allowed (default=False) multiedges: if True then multiple edges are allowed (default=False) Overrides: object.__init__

add_edge(*self*, *n1*, *n2*=None, *x*=None)

Add a single directed edge to the digraph.

Can be called as G.add_edge(n1,n2,x) or as G.add_edge(e), where e=(n1,n2,x).

If called as G.add_edge(n1,n2) or G.add_edge(e), with e=(n1,n2), then this is interpreted as adding the edge (n1,n2,None) to be compatible with the Graph and DiGraph classes.

n1,n2 are node objects, and are added to the Graph if not already present. Nodes must be hashable Python objects (except None).

x is an arbitrary (not necessarily hashable) object associated with this edge. It can be used to associate one or more, labels, data records, weights or any arbirary objects to edges. The default is the Python None.

For example, if the graph G was created with

```
>>> G=XDiGraph()
```

then G.add_edge(1,2,“blue”) will add the directed edge (1,2,“blue”).

If G.multiedges=False, then a subsequent G.add_edge(1,2,“red”) will change the above edge (1,2,“blue”) into the edge (1,2,“red”).

On the other hand, if G.multiedges=True, then two successive calls to G.add_edge(1,2,“red”) will result in 2 edges of the form (1,2,“red”) that can not be distinguished from one another.

If self.selfloops=False, then any attempt to create a self-loop with add_edge(n1,n1,x) will have no effect on the digraph and will not elicit a warning.

Objects imbedded in the edges from n1 to n2 (if any), can be retrieved using get_edge(n1,n2), or calling edges(n1) or edge_iter(n1) to return all edges attached to n1. Overrides: networkx.graph.Graph.add_edge

add_edges_from(*self*, *ebunch*)

Add multiple directed edges to the digraph. ebunch: Container of edges. Each edge e in container will be added using add_edge(e). See add_edge documentation. The container must be iterable or an iterator. It is iterated over once. Overrides: networkx.graph.Graph.add_edges_from

has_edge(*self*, *n1*, *n2*=None, *x*=None)

Return True if digraph contains directed edge (n1,n2,x).

Can be called as G.has_edge(n1,n2,x) or as G.has_edge(e), where e=(n1,n2,x).

If x is unspecified, i.e. if called with an edge of the form e=(n1,n2), then return True if there exists ANY edge from n1 to n2 (equivalent to has_successor(n1,n2)). Overrides: networkx.graph.Graph.has_edge

has_successor(*self*, *n1*, *n2*)

Return True if node n1 has a successor n2.

Return True if there exists ANY edge (n1,n2,x) for some x.

has_predecessor(*self*, *n1*, *n2*)

Return True if node n1 has a predecessor n2.

Return True if there exists ANY edge (n2,n1,x) for some x.

get_edge_iter(*self*, *u*, *v*=None)

Return an iterator over the objects associated with each edge from node u to node v.

get_edge(*self*, *u*, *v*=None)

Return the objects associated with each edge from node u to node v.

If multiedges=False, a single object is returned. If multiedges=True, a list of objects is returned. If no edge exists, None is returned. Overrides: networkx.graph.Graph.get_edge

delete_multiedge(*self*, *n1*, *n2*)

Delete all edges between nodes n1 and n2.

When there is only a single edge allowed between nodes (multiedges=False), this just calls delete_edge(n1,n2), otherwise (multiedges=True) all edges between n1 and n2 are deleted.

delete_edge(*self*, *n1*, *n2*=`None`, *x*=`None`, *all*=`False`)

Delete the directed edge (n1,n2,x) from the graph.

Can be called either as >>> G.delete_edge(n1,n2,x) or as >>> G.delete_edge(e) where e=(n1,n2,x).

If called with an edge e=(n1,n2), or as G.delete_edge(n1,n2) then the edge (n1,n2,None) will be deleted.

If the edge does not exist, do nothing.

To delete *all* edges between n1 and n2 use >>> G.delete_multiedges(n1,n2) Overrides: networkx.graph.Graph.delete_edge

delete_edges_from(*self*, *ebunch*)

Delete edges in ebunch from the graph.

ebunch: Container of edges. Each edge must be a 3-tuple (n1,n2,x) or a 2-tuple (n1,n2). The container must be iterable or an iterator, and is iterated over once.

Edges that are not in the graph are ignored. Overrides: networkx.graph.Graph.delete_edges_from

out_edges_iter(*self*, *nbunch*=`None`)

Return iterator that iterates once over each edge pointing out of nodes in nbunch, or over all edges in digraph if no nodes are specified.

See edges() for definition of nbunch.

Nodes in nbunch that are not in the graph will be (quietly) ignored. Overrides: networkx.digraph.DiGraph.out_edges_iter

in_edges_iter(*self*, *nbunch*=`None`)

Return iterator that iterates once over each edge pointing in to nodes in nbunch, or over all edges in digraph if no nodes are specified.

See edges() for definition of nbunch.

Nodes in nbunch that are not in the graph will be (quietly) ignored. Overrides: networkx.digraph.DiGraph.in_edges_iter

successors_iter(*self*, *n*)

Return an iterator of nodes pointing out of node n.

Returns the same data as out_edges(n) but in a different format. Overrides: networkx.digraph.DiGraph.successors_iter

predecessors_iter(*self*, *n*)

Return an iterator of nodes pointing in to node n.

Returns the same data as in_edges(n) but in a different format. Overrides: networkx.digraph.DiGraph.predecessors_iter

edges_iter(*self*, *nbunch*=`None`)

Return iterator that iterates once over each edge pointing out of nodes in nbunch, or over all edges in digraph if no nodes are specified.

See edges() for definition of nbunch.

Nodes in nbunch that are not in the graph will be (quietly) ignored. Overrides: networkx.graph.Graph.edges_iter

neighbors_iter(*self*, *n*)

Return an iterator of nodes pointing out of node n.

Returns the same data as out_edges(n) but in a different format. Overrides: networkx.graph.Graph.neighbors_iter

predecessors(*self*, *n*)

Return predecessor nodes of n. Overrides: networkx.digraph.DiGraph.predecessors

successors(*self*, *n*)

Return sucessor nodes of n. Overrides: networkx.digraph.DiGraph.successors

neighbors(*self*, *n*)

Return sucessor nodes of n. Overrides: networkx.graph.Graph.neighbors

in_degree_iter(*self*, *nbunch*=`None`, *with_labels*=`False`)

Return iterator for in_degree(n) or (n,in_degree(n)) for all n in nbunch.

If nbunch is ommitted, then iterate over *all* nodes.

See degree_iter method for more details. Overrides: networkx.digraph.DiGraph.in_degree_iter

out_degree_iter(*self*, *nbunch*=`None`, *with_labels*=`False`)

Return iterator for out_degree(n) or (n,out_degree(n)) for all n in nbunch.

If nbunch is ommitted, then iterate over *all* nodes.

See degree_iter method for more details. Overrides: networkx.digraph.DiGraph.out_degree_iter

degree_iter(*self*, *nbunch*=`None`, *with_labels*=`False`)

Return iterator that returns in_degree(n)+out_degree(n) or (n,in_degree(n)+out_degree(n)) for all n in nbunch. If nbunch is ommitted, then iterate over *all* nodes.

Can be called in three ways: G.degree_iter(n): return iterator the degree of node n G.degree_iter(nbunch): return a list of values, one for each n in nbunch (nbunch is any iterable container of nodes.) G.degree_iter(): same as nbunch = all nodes in graph.

If with_labels=True, iterator will return an (n,in_degree(n)+out_degree(n)) tuple of node and degree.

Any nodes in nbunch but not in the graph will be (quietly) ignored. Overrides: networkx.graph.Graph.degree_iter

nodes_with_selfloops(*self*)

Return list of all nodes having self-loops.

selfloop_edges(*self*)

Return all edges that are self-loops.

number_of_selfloops(*self*)

Return number of self-loops in graph.

allow_selfloops(*self*)

Henceforth allow addition of self-loops (edges from a node to itself).

This doesn't change the graph structure, only what you can do to it.

remove_all_selfloops(*self*)

Remove self-loops from the graph (edges from a node to itself).

ban_selfloops(*self*)

Remove self-loops from the graph and henceforth do not allow their creation.

allow_multiedges(*self*)

Henceforth allow addition of multiedges (more than one edge between two nodes).

Warning: This causes all edge data to be converted to lists.

remove_all_multiedges(*self*)

Remove multiedges retaining the data from the first edge

ban_multiedges(*self*)

Remove multiedges retaining the data from the first edge. Henceforth do not allow multiedges.

subgraph(*self*, *nbunch*, *inplace*=`False`, *create_using*=`None`)

Return the subgraph induced on nodes in nbunch.

nbunch: can be a single node or any iterable container of of nodes. (It can be an iterable or an iterator, e.g. a list, set, graph, file, numeric array, etc.)

Setting inplace=True will return induced subgraph in original graph by deleting nodes not in nbunch. It overrides any setting of create_using.

WARNING: specifying inplace=True makes it easy to destroy the graph.

Unless otherwise specified, return a new graph of the same type as self. Use (optional) create_using=R to return the resulting subgraph in R. R can be an existing graph-like object (to be emptied) or R can be a call to a graph object, e.g. create_using=DiGraph(). See documentation for empty_graph()

Note: use subgraph(G) rather than G.subgraph() to access the more general subgraph() function from the operators module. Overrides: networkx.graph.Graph.subgraph

copy(*self*)

Return a (shallow) copy of the digraph.

Return a new XDiGraph with same name and same attributes for selfloop and multiededges. Each node and each edge in original graph are added to the copy. Overrides: networkx.graph.Graph.copy

to_undirected(*self*)

Return the underlying graph of G.

The underlying graph is its undirected representation: each directed edge is replaced with an undirected edge.

If multiedges=True, then an XDiGraph with only two directed edges (1,2,"red") and (2,1,"blue") will be converted into an XGraph with two undirected edges (1,2,"red") and (1,2,"blue"). Two directed edges (1,2,"red") and (2,1,"red") will result in in two undirected edges (1,2,"red") and (1,2,"red").

If multiedges=False, then two directed edges (1,2,"red") and (2,1,"blue") can only result in one undirected edge, and there is no guarantee which one it is. Overrides: networkx.graph.Graph.to_undirected

reverse(*self*)

Return a new digraph with the same vertices and edges as self but with the directions of the edges reversed. Overrides: networkx.digraph.DiGraph.reverse

number_of_edges(*self*, *u*=`None`, *v*=`None`, *x*=`None`)

Return the number of edges between nodes u and v.

If u and v are not specified return the number of edges in the entire graph.

The edge argument e=(u,v) can be specified as G.number_of_edges(u,v) or G.number_of_edges(e) Overrides: networkx.graph.Graph.number_of_edges

Inherited from networkx.digraph.DiGraph(Section 9.2)

add_node(), add_nodes_from(), clear(), delete_node(), delete_nodes_from(), in_degree(), in_edges(), in_neighbors(), is_directed(), out_degree(), out_edges(), out_neighbors(), to_directed()

Inherited from networkx.graph.Graph(Section 28.2)

__contains__(), __getitem__(), __iter__(), __len__(), __str__(), add_cycle(), add_path(), degree(), edge_boundary(), edges(), has_neighbor(), has_node(), info(), node_boundary(), nodes(), nodes_iter(), number_of_nodes(), order(), prepare_nbunch(), size()

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __new__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __subclasshook__()

56.2.2 Properties

Name	Description
Inherited from object	
__class__	

57 Module networkx.xgraph

Base class for XGraph.

XGraph allows self-loops and multiple edges with arbitrary (hashable) objects as nodes and arbitrary objects associated with edges.

Examples Create an empty graph structure (a "null graph") with no nodes and no edges

```
>>> from networkx import *
>>> G=XGraph()  # default no self-loops, no multiple edges
```

You can add nodes in the same way as the simple Graph class >>> G.add_nodes_from(xrange(100,110))

You can add edges as for simple Graph class, but with optional edge data/labels/objects.

```
>>> G.add_edges_from([(1,2,0.776),(1,3,0.535)])
```

For graph coloring problems, one could use >>> G.add_edges_from([(1,2,"blue"),(1,3,"red")])
Author: Aric Hagberg (hagberg@lanl.gov) Pieter Swart (swart@lanl.gov) Dan Schult(dschult@colgate.edu

57.1 Variables

Name	Description
__package__	**Value:** `'networkx'`

57.2 Class XGraph

object

networkx.graph.Graph

networkx.xgraph.XGraph

A class implementing general undirected graphs, allowing (optional) self-loops, multiple edges, arbitrary (hashable) objects as nodes and arbitrary objects associated with edges.

An XGraph edge is specified by a 3-tuple e=(n1,n2,x), where n1 and n2 are nodes (hashable objects) and x is an arbitrary (and not necessarily unique) object associated with that edge.

```
>>> G=XGraph()
```

creates an empty simple and undirected graph (no self-loops or multiple edges allowed). It is equivalent to the expression:

```
>>> G=XGraph(name='',selfloops=False,multiedges=False)
```

```
>>> G=XGraph(name="empty",multiedges=True)
```

creates an empty graph with G.name="empty", that does not allow the addition of self-loops but does allow for multiple edges.

See also the XDiGraph class.

XGraph inherits from Graph, overriding the following methods:

- __init__
- add_edge
- add_edges_from
- has_edge, has_neighbor
- get_edge
- edges_iter
- delete_edge
- delete_edges_from
- degree_iter
- to_directed
- copy
- subgraph

XGraph adds the following methods to those of Graph:

- delete_multiedge
- nodes_with_selfloops
- selfloop_edges
- number_of_selfloops
- allow_selfloops
- remove_all_selfloops
- ban_selfloops
- allow_multiedges

- remove_all_multiedges
- ban_multiedges

57.2.1 Methods

__init__(*self*, *data*=None, *name*='', *selfloops*=False, *multiedges*=False)

Initialize XGraph.

Optional arguments:: name: graph name (default=") selfloops: if True selfloops are allowed (default=False) multiedges: if True multiple edges are allowed (default=False) Overrides: object.__init__

__getitem__(*self*, *n*)

Return the neighbors of node n as a list.

This provides graph G the natural property that G[n] returns the neighbors of G. Overrides: networkx.graph.Graph.__getitem__

add_edge(*self*, *n1*, *n2*=None, *x*=None)

Add a single edge to the graph.

Can be called as G.add_edge(n1,n2,x) or as G.add_edge(e), where e=(n1,n2,x).

n1,n2 are node objects, and are added to the Graph if not already present. Nodes must be hashable Python objects (except None).

x is an arbitrary (not necessarily hashable) object associated with this edge. It can be used to associate one or more: labels, data records, weights or any arbirary objects to edges. The default is the Python None.

For example, if the graph G was created with

```
>>> G=XGraph()
```

then G.add_edge(1,2,“blue”) will add the edge (1,2,“blue”).

If G.multiedges=False, then a subsequent G.add_edge(1,2,“red”) will change the above edge (1,2,“blue”) into the edge (1,2,“red”).

If G.multiedges=True, then two successive calls to G.add_edge(1,2,“red”) will result in 2 edges of the form (1,2,“red”) that can not be distinguished from one another.

G.add_edge(1,2,“green”) will add both edges (1,2,X) and (2,1,X).

If self.selfloops=False, then calling add_edge(n1,n1,x) will have no effect on the Graph.

Objects associated to an edge can be retrieved using edges(), edge_iter(), or get_edge(). Overrides: networkx.graph.Graph.add_edge

add_edges_from(*self*, *ebunch*)

Add multiple edges to the graph.

ebunch: Container of edges. Each edge must be a 3-tuple (n1,n2,x) or a 2-tuple (n1,n2). See add_edge documentation.

The container must be iterable or an iterator. It is iterated over once. Overrides: networkx.graph.Graph.add_edges_from

has_edge(*self*, *n1*, *n2*=None, *x*=None)

Return True if graph contains edge (n1,n2,x).

Can be called as G.has_edge(n1,n2,x) or as G.has_edge(e), where e=(n1,n2,x).

If x is unspecified or None, i.e. if called with an edge of the form e=(n1,n2), then return True if there exists ANY edge between n1 and n2 (equivalent to has_neighbor(n1,n2)) Overrides: networkx.graph.Graph.has_edge

has_neighbor(*self*, *n1*, *n2*)

Return True if node n1 has neighbor n2.

Note that this returns True if there exists ANY edge (n1,n2,x) for some x. Overrides: networkx.graph.Graph.has_neighbor

neighbors_iter(*self*, *n*)

Return an iterator of nodes connected to node n.

Returns the same data as edges(n) but in a different format. Overrides: networkx.graph.Graph.neighbors_iter

neighbors(*self*, *n*)

Return a list of nodes connected to node n. Overrides: networkx.graph.Graph.neighbors

get_edge_iter(*self*, *u*, *v*)

Return an iterator over the objects associated with each edge from node u to node v.

get_edge(*self*, *u*, *v*)

Return the objects associated with each edge from node u to node v.

If multiedges=False, a single object is returned. If multiedges=True, a list of objects is returned. If no edge exists, None is returned. Overrides: networkx.graph.Graph.get_edge

edges_iter(*self*, *nbunch*=`None`)

Return iterator that iterates once over each edge adjacent to nodes in nbunch, or over all nodes in graph if nbunch=None.

If nbunch is None return all edges in the graph. The argument nbunch can be any single node, or any sequence or iterator of nodes. Nodes in nbunch that are not in the graph will be (quietly) ignored. Overrides: networkx.graph.Graph.edges_iter

delete_multiedge(*self*, *n1*, *n2*)

Delete all edges between nodes n1 and n2.

When there is only a single edge allowed between nodes (multiedges=False), this just calls delete_edge(n1,n2) otherwise (multiedges=True) all edges between n1 and n2 are deleted.

delete_edge(*self*, *n1*, *n2*=None, *x*=None)

Delete the edge (n1,n2,x) from the graph.

Can be called either as

```
>>> G.delete_edge(n1,n2,x)
```

or

```
>>> G.delete_edge(e)
```

where e=(n1,n2,x).

The default edge data is x=None

If called with an edge e=(n1,n2), or as G.delete_edge(n1,n2) then the edge (n1,n2,None) will be deleted.

If the edge does not exist, do nothing.

To delete *all* edges between n1 and n2 use >>> G.delete_multiedges(n1,n2)
Overrides: networkx.graph.Graph.delete_edge

delete_edges_from(*self*, *ebunch*)

Delete edges in ebunch from the graph.

ebunch: Container of edges. Each edge must be a 3-tuple (n1,n2,x) or a 2-tuple (n1,n2). In the latter case all edges between n1 and n2 will be deleted. See delete_edge.

The container must be iterable or an iterator, and is iterated over once. Edges that are not in the graph are ignored. Overrides: networkx.graph.Graph.delete_edges_from

degree_iter(*self*, *nbunch*=None, *with_labels*=False)

This is the degree() method returned in iterator form. If with_labels=True, iterator yields 2-tuples of form (n,degree(n)) (like iteritems() on a dict.)
Overrides: networkx.graph.Graph.degree_iter

copy(*self*)

Return a (shallow) copy of the graph.

Return a new XGraph with same name and same attributes for selfloop and multiededges. Each node and each edge in original graph are added to the copy. Overrides: networkx.graph.Graph.copy

to_directed(*self*)

Return a directed representation of the XGraph G.

A new XDigraph is returned with the same name, same nodes and with each edge (u,v,x) replaced by two directed edges (u,v,x) and (v,u,x). Overrides: networkx.graph.Graph.to_directed

nodes_with_selfloops(*self*)

Return list of all nodes having self-loops.

selfloop_edges(*self*)

Return all edges that are self-loops.

number_of_selfloops(*self*)

Return number of self-loops in graph.

allow_selfloops(*self*)

Henceforth allow addition of self-loops (edges from a node to itself).

This doesn't change the graph structure, only what you can do to it.

remove_all_selfloops(*self*)

Remove self-loops from the graph (edges from a node to itself).

ban_selfloops(*self*)

Remove self-loops from the graph and henceforth do not allow their creation.

allow_multiedges(*self*)

Henceforth allow addition of multiedges (more than one edge between two nodes).

Warning: This causes all edge data to be converted to lists.

remove_all_multiedges(*self*)

Remove multiedges retaining the data from the first edge

ban_multiedges(*self*)

Remove multiedges retaining the data from the first edge. Henceforth do not allow multiedges.

subgraph(*self*, *nbunch*, *inplace*=`False`, *create_using*=`None`)

Return the subgraph induced on nodes in nbunch.

nbunch: can be a single node or any iterable container of of nodes. (It can be an iterable or an iterator, e.g. a list, set, graph, file, numeric array, etc.)

Setting inplace=True will return induced subgraph in original graph by deleting nodes not in nbunch. It overrides any setting of create_using.

WARNING: specifying inplace=True makes it easy to destroy the graph.

Unless otherwise specified, return a new graph of the same type as self. Use (optional) create_using=R to return the resulting subgraph in R. R can be an existing graph-like object (to be emptied) or R can be a call to a graph object, e.g. create_using=DiGraph(). See documentation for empty_graph()

Note: use subgraph(G) rather than G.subgraph() to access the more general subgraph() function from the operators module. Overrides: networkx.graph.Graph.subgraph

number_of_edges(*self*, *u*=`None`, *v*=`None`, *x*=`None`)

Return the number of edges between nodes u and v.

If u and v are not specified return the number of edges in the entire graph.

The edge argument e=(u,v) can be specified as G.number_of_edges(u,v) or G.number_of_edges(e) Overrides: networkx.graph.Graph.number_of_edges

Inherited from networkx.graph.Graph(Section 28.2)

__contains__(), __iter__(), __len__(), __str__(), add_cycle(), add_node(), add_nodes_from(), add_path(), clear(), degree(), delete_node(), delete_nodes_from(), edge_boundary(), edges(), has_node(), info(), is_directed(), node_boundary(), nodes(), nodes_iter(), number_of_nodes(), order(), prepare_nbunch(), size(), to_undirected()

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __new__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __subclasshook__()

57.2.2 Properties

Name	Description
Inherited from object	
__class__	